MARL
Multi Agent Reinforcement Learning

Report finale per il corso di Sistemi Distribuiti - LM Ingegneria e Scienze Informatiche

veronika.folin@studio.unibo.it
davide.domini@studio.unibo.it

Ottobre 2022

Al giorno d'oggi stiamo assistendo ad un rapido sviluppo nei campi del *Pervasive Computing* e dell'*Intelligenza Artificiale*. Quotidianamente siamo circondati da oggetti e/o sistemi intelligenti, sviluppati per aiutarci nei compiti da svolgere. Questo può avvenire, ad esempio, nella nostra abitazione mediante infissi, o elettrodomestici smart oppure sul posto di lavoro dove possono essere presenti sistemi robotici applicati alle catene di montaggio o al comparto logistica. Il campo del *Multi Agent Reinforcement Learning* cerca di unire le due discipline sopra citate, al fine di rendere sempre più autonomi i sistemi che ci circondano. Questo progetto ha lo scopo di effettuare una panoramica del paradigma MARL, producendo una sezione teorica e una sezione pratica dove vengono testate due delle tecnologie più promettenti e stabili in questo ambito: PettingZoo e Mava. A tal fine, sono stati condotti diversi esperimenti che riproducono alcuni dei più classici problemi nel campo MARL e ne sono stati riportati i risultati ottenuti, evidenziando le potenzialità e i limiti di questi framework.

Contents

List of Figures

1 Paradigma ad agenti

1.1 Introduzione: dagli oggetti agli agenti

Nel corso degli anni si è verificata una crescita in termini di complessità dei sistemi informatici prodotti: ciò ha portato allo sviluppo di diversi paradigmi di programmazione che permettessero di produrre codice in modo più agevole e, allo stesso tempo, sempre più articolato.

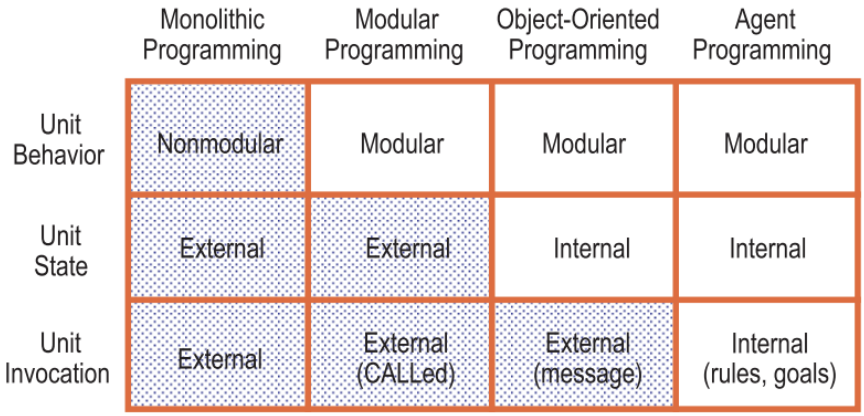


Figure 1: Evoluzione dei paradigmi di programmazione. [12]

Per capire le motivazioni che hanno portato allo sviluppo del paradigma ad agenti bisogna porsi principalmente due domande:

1. Quali sono le nuove caratteristiche dei sistemi complessi che vogliamo sviluppare?
2. Perché il paradigma ad oggetti non modella in maniera esaustiva questi sistemi? Quali limiti ha?

Caratteristiche dei nuovi sistemi complessi

Cercando di rispondere alla prima domanda, possiamo riassumere le caratteristiche dei moderni sistemi complessi distribuiti nel seguente modo. Questi sistemi sono:

- **diffusi**

  Parliamo di sistemi embedded che possono essere integrati nella maggior parte degli oggetti che ci circondano (e.g. frigoriferi, macchine, smart-watch, televisori, ecc...).

- **sempre connessi**

  Grazie alle tecnologie wireless questi sistemi hanno una connessione pervasiva.

- **sempre attivi**

Possono svolgere dei compiti al nostro posto (e.g. braccio robotico che aiuta un operaio in fabbrica).

- **sempre più autonomi**

Possiamo dunque estrapolare quattro feature principali di questi sistemi:

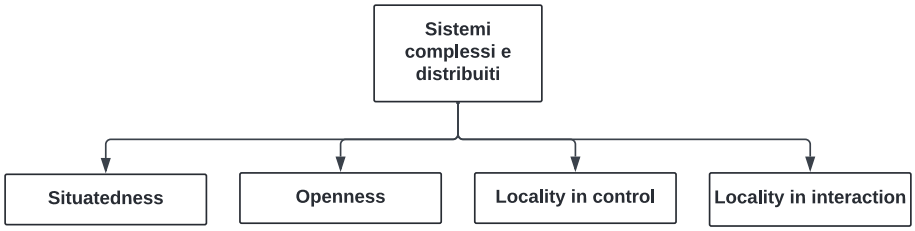


Figure 2: Feature chiave dei moderni sistemi complessi e distribuiti

1. ***Situatedness***: la computazione avviene in un *environment*, ovvero il contesto di esecuzione di un dato agente. Anche nel paradigma ad oggetti sono presenti i contesti di esecuzione ma, in questo caso, non vengono espressi come concetti di prima classe e ciò ne limita l'espressività.

2. ***Openness***: questi sistemi devono essere aperti al cambiamento rispetto la cardinalità, l'organizzazione, la struttura e sotto molti altri aspetti.

3. ***Locality in control***: ogni agente possiede il controllo sulle proprie azioni e queste vengono intraprese localmente. L'astrazione utilizzata è quella dell'*autonomia* e permette di disaccoppiare le varie entità, oltre che semplificare il movimento degli agenti da un environment ad un altro.

4. ***Locality in interaction***: gli agenti autonomi interagiscono con l'ambiente in cui sono immersi, inoltre questa interazione può essere limitata ad una sottoparte circoscritta dell'ambiente mediante leggi fisiche o vincoli logici.

Limiti del paradigma ad oggetti

Per quanto riguarda la seconda domanda, invece, ci sono diversi punti da considerare:

- il flusso di controllo è centralizzato a *design-time*;
- i sistemi che possiamo costruire con il paradigma OO sono predicibili;
- i sistemi OO non hanno una corrispondenza con un elemento della natura; contrariamente, le analogie con la natura e/o con le società hanno aiutato lo sviluppo di sistemi artificiali.

Questi aspetti rappresentano degli over-constraints che rendono complicato e non scalabile lo sviluppo di sistemi distribuiti complessi tramite l'OOP.

1.2 Key feature

Per capire effettivamente cos'è un agente possiamo partire dalla sua definizione:

"Un agente è un'entità computazionale autonoma che incapsula il suo flusso di controllo e un criterio per auto-governarsi."

Questa introduce diversi punti di discussione:

- gli agenti sono autonomi e questo implica che:
  - non possono essere invocati dall'esterno, non hanno un'interfaccia pubblica;
  - sono impredicibili, o parzialmente-predicibili.
- il flusso di controllo è incapsulato all'interno degli agenti e questi comunicano tra loro attraverso flussi di dati. Ciò implica che possiamo progettare i sistemi dal punto di vista di questo flusso;
- il fatto che gli agenti possano interagire fra loro implica che questi possano vivere in un MAS (Multi-Agent System);
- gli agenti sono immersi in un ambiente con il quale interagiscono.

Queste osservazioni ci conducono alla definizione di ***weak-agent***:

Un agente è detto weak quando ha le seguenti quattro caratteristiche:

1. ***Autonomia***
2. ***Proattività***
3. ***Reattività ai cambiamenti***
4. ***Socialità***

La prima caratteristica, ovvero l'autonomia, può essere interpretata in svariati modi, dai più semplici ai più complessi. *Implica la necessità per un agente di essere intelligente o di essere capace di imparare?* In realtà no: nella sua forma più basica un agente può essere un automa a stati finiti con un proprio flusso di controllo e quindi, per questo, viene definito anche weak-agent. Andando ad estendere la definizione base di autonomia si possono introdurre dei concetti, non strettamente necessari in tutti i contesti, che vanno ad incrementare il grado di utilità dell'agente stesso. Ad esempio:

- ***Intelligenza***: rende più semplice il self-government dell'agente stesso;
- ***Capacità di imparare***: un agente potrebbe imparare a svolgere nuovi compiti o incrementare le proprie capacità di ragionamento;
- ***Mobilità***: indica che il flusso di controllo dell'agente è indipendente dall'environment in cui vive.

Questa estensione ci porta alla definizione degli ***strong-agents***:

"Gli strong-agents hanno dei componenti come: desideri, goal, piani, conoscenza del mondo, ..."

2 Reinforcement Learning

2.1 Overview

Il *reinforcement learning* è una tecnica di apprendimento automatico che permette ad un agente di scegliere in autonomia le azioni da compiere; tali scelte vengono influenzate dallo stato dell'ambiente in cui è immerso e con cui può interagire.

A differenza dei metodi di *learning supervisionato*, il *reinforcement learning* non si basa su un dataset di esempi etichettati, in quanto è difficile ottenere un insieme rappresentativo dei comportamenti desiderabili per l'agente. Inoltre, l'*apprendimento per rinforzo* si distingue dal paradigma *non supervisionato* in quanto ha l'obiettivo di massimizzare un segnale di reward, al posto di trovare un pattern nascosto nei dati. Difatti all'agente non viene detto quale azione intraprendere ma dovrà calcolare, esplorando l'insieme delle azioni possibili, quella che gli porterà ad avere un reward maggiore. Ottenendo un reward ad ogni step dell'algoritmo, l'agente svilupperà un certo grado di esperienza che gli permetterà di apprendere l'insieme di azioni ottimali.

Il paradigma di reinforcement learning si basa su quattro concetti:

- ***policy***, definisce il modo con cui l'agente apprende e il suo comportamento in un dato istante. Nei casi più semplici può essere espressa come una *funzione di lookup*, altrimenti può essere implementata con processi di ricerca più avanzati. Inoltre, possiamo distinguere le policy in *deterministiche*, ossia che dipendono dal solo stato, oppure in *stocastiche*, ovvero definite mediante distribuzioni di probabilità sulle azioni dato uno stato.

- ***reward***, definisce il goal di un problema attraverso l'invio di un segnale che indica la bontà di un'azione. Questo permette di alterare in modo appropriato la policy: se viene ricevuto un reward basso, la policy verrà modificata affinchè selezioni un'azione diversa in futuro.

- ***funzione valore***, specifica un reward a lungo termine (o ***valore di ritorno***), ovvero l'ammontare totale dei reward che un agente si aspetta di accumulare nel futuro, partendo da quello stato. In generale, l'algoritmo dovrà scegliere le azioni che massimizzano la funzione valore, non la ricompensa nell'immediato, modificando opportunamente la policy.

- ***modello dell'ambiente***, ovvero un'entità in grado di simulare il comportamento dell'ambiente.

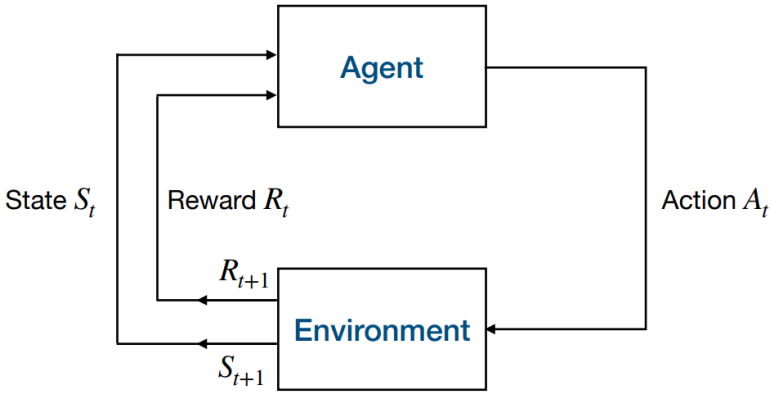


Figure 3: Funzionamento dell'apprendimento per rinforzo

2.2 Markov Decision Process

Nel reinforcement learning un ambiente può essere descritto formalmente attraverso un *Markov Decision Process* che comprende:

- un insieme finito di **stati** S;
- un insieme finito di **azioni** A;
- una **funzione di transizione** T che assegna ad ogni coppia stato-azione una distribuzione di probabilità su S;
- una **funzione di rinforzo (o *reward*)** R che assegna un valore numerico ad ogni possibile transizione.

La funzione di transizione e di rinforzo non sono necessariamente note all'agente.

L'ipotesi Markoviana implica che quello che succede all'istante $t + 1$ dipende solo da quello che è successo all'istante precedente t e non da tutto ciò che è successo in precedenza. Se un problema soddisfa questa ipotesi, allora sarà possibile trattarlo mediante ***episodi***, ovvero tentativi ripetuti di raggiungere un determinato obiettivo.

2.3 Soluzioni model-based Vs model-free

Oltre alla funzione valore, ci sono altri modi per misurare la bontà di un'azione in un certo stato:

- **funzione azione-valore** (o *Q-function*), che viene definita come la somma dei rewards prevista quando si intraprende un'azione in un determinato stato, seguendo una certa policy;

- **equazione di Bellman**, che formula il valore della soluzione ottimale del problema (massimizzare la somma dei reward attesa) in termini di programmazione dinamica, ovvero scomponendolo in una sequenza di sottoproblemi concatenati. Questo approccio è applicabile nel caso siano conosciute le probabilità di transizione e le funzioni di reward, tali algoritmi sono conosciuti come ***model-based***.

D'altra parte, esistono algoritmi per cui non è richiesto alcun modello dell'ambiente o per cui il sistema corrispondente è troppo complesso da descrivere (es: un gioco con un grandissimo numero di configurazioni). In questi casi si assume che le probabilità non siano conosciute, che la policy e la funzione valore debbano essere stimate attraverso uno o più step di simulazione, al fine di testare i possibili scenari. Tali metodi sono conosciuti come algoritmi ***model-free***: i più utilizzati sono *Monte Carlo*, *Temporal Difference* e *Policy Search*.

3 MARL

3.1 Overview

Un sistema multi-agente è formato da molteplici entità che, con un certo grado di autonomia, prendono decisioni ed interagiscono fra loro all'interno di un ambiente [16]. Ad ogni agente viene assegnato un compito da perseguire, questo può essere uguale a quello degli altri (cooperativo) oppure diverso (competitivo). Un dato agente per risolvere un determinato task ha bisogno di avere un insieme di skills: esprimere queste competenze in modo programmatico sarebbe troppo complesso, se non impossibile. Per questo motivo si è deciso di utilizzare il Reinforcement Learning e da qui nasce il termine *Multi-Agent Reinforcement Learning*, di cui possiamo dare la seguente definizione:

*"Un gruppo di agenti **impara insieme** la miglior policy per massimizzare una funzione di reward a lungo termine."*

3.1.1 Environment

Nel dominio del MARL, l'environment ha due caratteristiche principali:

1. ***Osservabilità parziale***, per cui un dato agente non ha una rappresentazione completa dell'ambiente in cui vive;
2. ***Non-stazionarità***, per cui l'ambiente cambia nel tempo.

Queste due caratteristiche portano una complicazione rispetto al *Single Agent Reinforcement Learning*. Difatti, nel SARL il processo di apprendimento è modellato come un Markov Decision Process ma questo modello è stazionario per definizione. Questo vincolo porta ad avere obbligatoriamente un solo agente che effettua il learning e a considerare i restanti agenti come parte dell'environment stesso. Nel caso del MARL, dunque, il Reinforcement Learning viene integrato utilizzando un'ulteriore astrazione: ***Stochastic Game $\mathcal{S}$ (o Markov Games)*** [10]:

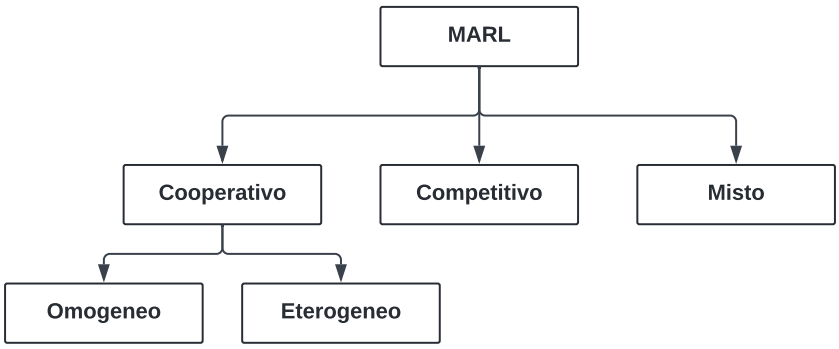
- $\mathcal{S}$ è una tupla $< N, S, \{A^i\}, P, \{R^i\} >$ con $i \in 1..N$
- N è il numero di agenti (deve essere maggiore di 1 ma non ha un upper-bound)
- S è lo stato globale dell'environment
- A^i è lo spazio delle azioni del i-esimo agente
  - Lo spazio globale viene definito come $\mathbb{A} = A^1 \times A^2 \times .... \times A^N$
- $P : S \times \mathbb{A} \to \mathcal{P}(S)$ è la funzione che descrive i passaggi fra uno stato e l'altro
  - $P(S)$ è una distribuzione di probabilità discreta che associa ad ogni stato $s \in S$ una probabilità
- $R^i : S \times \mathbb{A} \times S \to \mathbb{R}$ è la funzione di reward

Esempio di Markov Game: Paper, Rock and Scissor

- $N = 2$
- $A^1 = A^2 = \{\text{Paper, Rock, Scissor}\}$
- $S = \{\ \}$
- $R = \begin{array}{c} \\ Rock \\ Paper \\ Scissor \end{array} \begin{array}{c} \begin{array}{ccc} Rock & Paper & Scissor \end{array} \\ \begin{bmatrix} 0,0 & -1,1 & 1,-1 \\ 1,-1 & 0,0 & -1,1 \\ -1,1 & 1,-1 & 0,0 \end{bmatrix} \end{array}$

3.1.2 Tipi di MARL

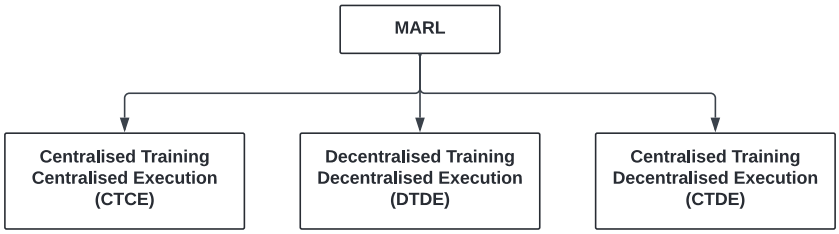
I sistemi MARL possono essere categorizzati in diversi modi, una prima divisione è la seguente:



- ***Cooperativo***: tutti gli agenti presenti nel sistema hanno il medesimo goal, quindi, anche la medesima funzione di reward ($R^1 = ... = R^i = ... = R^N$).
  - ***Omogeneo***: tutti gli agenti presenti nel sistema hanno le medesime capacità ($A^1 = ... = A^i = ... = A^N$). Lo scopo globale è, dunque, trovare la miglior policy, che sarà la medesima per ogni agente ($\pi^* = \pi_1^* = ... = \pi_N^*$).
  - ***Eterogeneo***: gli agenti presenti nel sistema possono avere capacità diverse e nel caso peggiore ogni agente ha capacità diverse da tutti gli altri ($A^1 \neq ... \neq A^i \neq ... \neq A^N$). Lo scopo globale è, dunque, massimizzare la policy locale di ogni agente, seguendo il goal condiviso.
- ***Competitivo***: gli agenti presenti nel sistema hanno goal diversi e competono fra loro per massimizzare la propria funzione di reward. Un esempio tipico di questi sistemi è rappresentato dai giochi da tavolo.

- ***Misto***: in questo tipo di sistemi sono presenti sia agenti che condividono lo stesso goal (cooperativi) sia agenti che hanno goal diversi (competitivi). Un esempio classico sono i giochi a squadre dove i giocatori di una stessa squadra cooperano mentre i giocatori di due squadre diverse competono.

È possibile fare un'ulteriore suddivisione, rispetto a come avviene il training e l'esecuzione dei vari agenti:



- ***Centralised Training Centralised Execution***: in questo tipo di sistemi è presente un agente *Learner*, di livello "superiore" e con una visione globale, il cui compito è effettuare il training e computare le azioni da intraprendere. I restanti agenti, dunque, invieranno quello che percepiscono dall'environment (stato locale) all'agente Learner, che si occuperà di effettuare un'unione per ricostruire lo stato globale. Una volta ricostruito, sceglie l'azione da intraprendere, in accordo con la policy corrente, e valuta la funzione di reward per aggiornare la policy (processo di learning). Solitamente questo approccio è usato per effettuare il training offline del sistema.

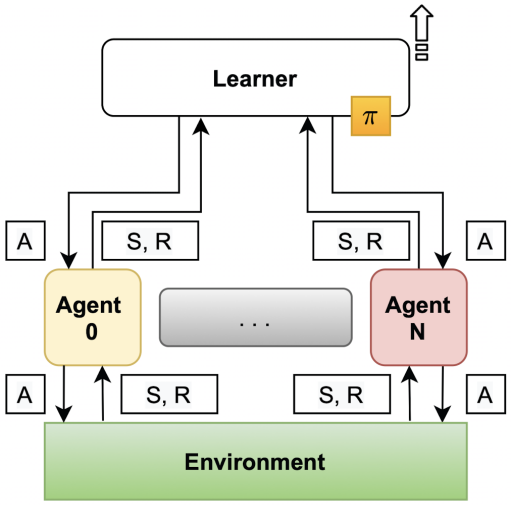


Figure 4: CTCE System [1]

- ***Decentralised Training decentralised Execution***: in questo tipo di sistemi ogni agente ha una sua policy locale e può osservare solo una porzione dell'envirnoment.

Ogni agente, dunque, intraprenderà delle azioni basate sulla propria percezione dell'environment e aggiornerà di conseguenza la propria funzione di reward locale. Questo approccio è utilizzato sia per il training offline che per il deployment del sistema (online).

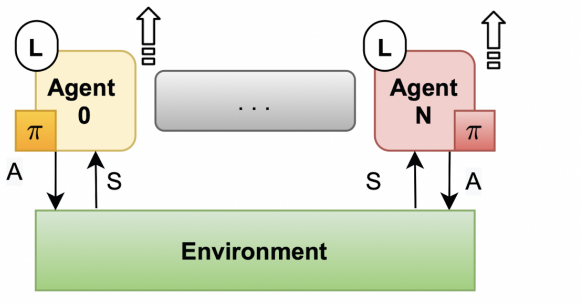


Figure 5: DTDE System [1]

- ***Centralised Training Decentralised Execution***, in questo tipo di sistemi sono presenti due step:
  1. ***Simulation Time***: ogni agente utilizza la propria policy locale e alla fine di ogni episodio invia le rispettive informazioni ad un agente Learner centrale il quale, avendo una visione globale dell'environment, aggiornerà le policy locali dei vari agenti.
  2. ***Execution Time***: in questa fase ogni agente utilizza la propria policy locale (risultato del processo di training) per agire sull'environment.

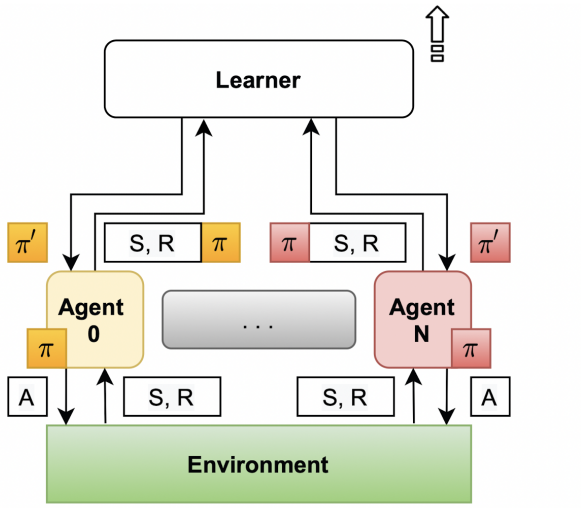


Figure 6: CTDE System [1]

3.2 Challenges

In questa sezione sono descritte le sifde in ambito MARL, che sono ancora oggetto di ricerca.

Environment parzialmente osservabili In environment di una certa complessità (quindi escludendo i toy-environment) gli agenti non possono avere una rappresentazione completa del mondo che li circonda nè possono avere accesso allo stato degli altri agenti presenti. Questo porta l'environment a non soddisfare la condizione di Markov.

Non-Stazionarità Questo problema deriva dalla presenza di molteplici agenti che cercano di imparare contemporaneamente: dal punto di vista di un singolo agente, l'environment diventa non-stazionario. Questo porta a violare il vincolo di Markov e ne deriva che il learning diventa molto complesso. [7] Una possibile soluzione, descritta in precedenza, è data dall'astrazione dei Markov Games.

Comunicazione La comunicazione fra agenti è una feature ritenuta molto importante in letteratura. In particolare, il problema non affronta solo il cosa un agente deve comunicare ma anche con chi deve comunicare e quando. Questo aspetto è stato ampiamente studiato, e lo è tutt'ora, in quanto ha un impatto sulle restanti problematiche descritte in questa sezione [6]:

- *Non-stazionarità*: se un agente può ricevere comunicazione dagli altri agenti rispetto le azioni da loro intraprese allora, dal suo punto di vista, il problema torna ad essere stazionario;
- *Coordinazione*: lo scambio di informazioni fra i vari agenti è fondamentale per poter raggiungere il consenso su un'azione da intraprendere;
- *Osservabilità parziale*: la possibilità di scambiare comunicazioni permette agli agenti di scambiare la propria percezione dell'environment e, quindi, di ricostruire uno stato globale.

Coordinazione La coordinazione è una feature essenziale nei sistemi cooperativi, in quanto è necessario che gli agenti raggiungano un consenso sulle azioni da intraprendere. Se questo non avviene in fase di learning, si può arrivare a scegliere una policy subottima. La coordinazione può essere ottenuta mediante comunicazione, come descritto nella sezione precedente, oppure implicitamente: ogni agente costruisce un proprio modello dei restanti agenti, cercando di comprendere i pattern comportamentali e inferire la prossima azione.

Credit Assignment Problem Il Credit Assignment Problem si riferisce al problema di associare un reward ad un'azione intrapresa da un dato agente. Questa associazione è necessaria, in fase di learning, per capire l'efficacia di un'azione intrapresa e riuscire a massimizzare la funzione di reward nel tempo. Alcuni studi [5] mostrano come, in

caso di agenti indipendenti, questi non siano in grado di distinguere gli effetti derivanti dal comportamento stocastico dell'environment dalle azioni intraprese dagli altri agenti. Inoltre, la complessità di questo problema aumenta considerando due ulteriori vincoli:

1. la natura del Reinforcement Learning, per cui, non si deve capire solo l'impatto di una singola azione ma di una sequenza di azioni nel tempo;
2. la parziale osservabilità degli environment.

Scalabilità La scalabilità di questi sistemi è influenzata dai problemi descritti precedentemente in questa sezione. Effettuare il training di un agente è di per sé difficile: aumentando il numero di agenti del sistema, questa complessità aumenta esponenzialmente. Oltretutto, la scalabilità del sistema è determinata anche dalla robustezza di un agente a fronte di un cambiamento nel comportamento degli altri agenti. In letteratura sono presenti alcune tecniche, in fase di esplorazione, utili a rafforzare la scalabilità come, ad esempio, knowledge reuse, regolarizzazione, adversarial training ed altre ancora.

3.3 Esempi di successo

OpenAI "Hide and Seek"

Nell'articolo *Emergent Tool Use From Multi-Agent Autocurricula* [2], realizzato dal gruppo di ricerca OpenAI, vengono presentati degli esperimenti condotti su un sistema ad agenti, i quali sono suddivisi in due tipologie in base al tipo di goal: *hiders* (coloro che si devono nascondere) e *seekers* (coloro che devono trovare gli agenti nascosti). Questi si trovano all'interno di un environment in cui vengono posizionati randomicamente oggetti ed ostacoli ad ogni iterazione.

Gli agenti osservati sono stati in grado di congegnare strategie e controstrategie, anche inaspettate, per raggiungere il proprio obiettivo: principalmente attraverso l'interazione con gli oggetti presenti, nonostante non fosse stato fissato nessun tipo di incentivo per gli agenti affinché li utilizzassero. Tale risultato è definito ***autocurriculum*** ed è stato indotto meramente dalla competizione fra gli agenti. Si può osservare, infatti, che ogni nuova ***behaviour*** appresa dagli agenti è una diretta strategia di difesa/attacco rispetto a quelle intraprese dagli avversari. Per esempio, nel caso di un environment semplice, gli esperimenti condotti hanno ottenuto progressivamente i seguenti risultati:

1. gli agenti si muovono randomicamente;
2. i *seekers* imparano a inseguire gli *hiders*;
3. gli *hiders* muovono gli oggetti per bloccare il passaggio agli avversari;
4. i *seekers* utilizzano le rampe per saltare gli ostacoli;
5. gli *hiders* nascondono le rampe all'interno della stanza affinchè i *seekers* non possano utilizzarle;

6. gli *hiders* imparano a coordinarsi affinché alcuni di loro si occupino di bloccare le entrate e altri si occupino di nascondere le rampe.

Nel caso di un environment più complesso, in cui la cardinalità degli elementi che lo compongono può variare in un range più ampio, gli agenti hanno avuto bisogno di effettuare molte più partite per sviluppare delle strategie di gioco. La maggior complessità dell'environment ha permesso, però, di congegnare abilità anche diverse dai primi esperimenti, come:

1. gli *hiders* hanno imparato a costruire un rifugio per nascondersi all'interno;
2. gli *hiders* hanno imparato a bloccare le rampe per impedire ai *seekers* di utilizzarle;
3. i *seekers* hanno imparato a salire sui box e spostarli affinché possano affacciarsi sul rifugio creato dagli *hiders*;
4. gli *hiders* hanno imparato a bloccare tutte le rampe e i box per impedirne l'utilizzo ai *seekers*.

Per gli esperimenti effettuati è stata sfruttata un'architettura ispirata al modello *Transformer*. Tuttavia, esiste un'altra tecnica non supervisionata applicabile al problema in esame ed è la cosiddetta ***intrinsic motivation***, ovvero un metodo di learning che incentiva gli agenti ad esplorare l'environment, in base ad alcune metriche come il *model error* o lo *state count*. Alla luce di ciò sono stati confrontati il modello *multi agent competition* con quello *count-based* ed è stato evidenziato come quest'ultimo non ottenesse risultati altrettanto soddisfacenti: gli agenti, all'aumentare della complessità, hanno interazioni sempre meno significative con gli oggetti che li circondano.

Al fine di testare le capacità apprese dal sistema ad agenti, sono stati sviluppati dei ***domain-specific intelligence test***, ovvero dei test che propongono dei nuovi goal agli agenti addestrati. Gli esperimenti hanno ottenuto i seguenti risultati:

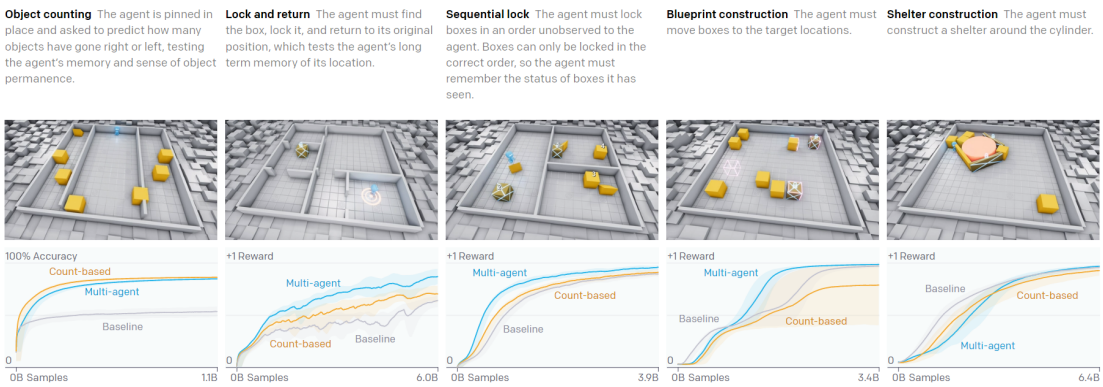


Figure 7: Hide and Seek: domain-specific intelligence test su 3 diversi sistemi ad agenti

DeepMind "Capture the flag"

Anche nel lavoro pubblicato [9] dal gruppo di ricerca DeepMind vengono evidenziati risultati sorprendenti riguardo l'applicazione di un sistema multi-agente. Difatti, come dimostrato dall'esempio riportato precedentemente, il successo di questi sistemi deriva dall'abilità di sviluppare, "naturalmente" e in maniera non supervisionata, un ***curriculum di behaviour*** nel corso dell'addestramento. Le abilità acquisite sono comparabili o, in molti casi, superiori a quelle umane.

Per esempio, nell'articolo preso in esame, viene testato un sistema ad agenti con la seguente logica:

- vi sono due team che competono in una mappa di gioco e hanno l'obiettivo di rubare la bandiera dell'avversario e di proteggere la propria;
- ogni concorrente può toccare un membro della squadra avversaria per riportarlo al punto di partenza;
- il team con più bandiere rubate, dopo cinque minuti, vince
- i giocatori devono cooperare con i propri compagni di squadre e, allo stesso tempo, competere con gli avversari;
- la mappa di gioco cambia ad ogni match ed in questo modo i giocatori devono acquisire delle strategie più che memorizzare un percorso ottimale;
- ogni concorrente riceve un segnale di reinforcement learning per match che gli indica se la propria squadra ha vinto o meno;
- gli agenti operano rispetto due scale temporali (veloce o lenta), al fine di ottimizzare l'uso della memoria e generare sequenze di azioni.

Le policy apprese mediante l'addestramento sono risultate robuste rispetto alle dimensioni della mappa, al numero di compagni di squadra e al numero di avversari.

In aggiunta ai precedenti esperimenti, ne sono stati condotti altri in cui veniva chiesto a degli esseri umani di giocare all'interno di squadre miste (agenti + altre persone). Quest'ultimo sistema è diventato più forte di qualsiasi altro sistema *baseline* e degli stessi umani.

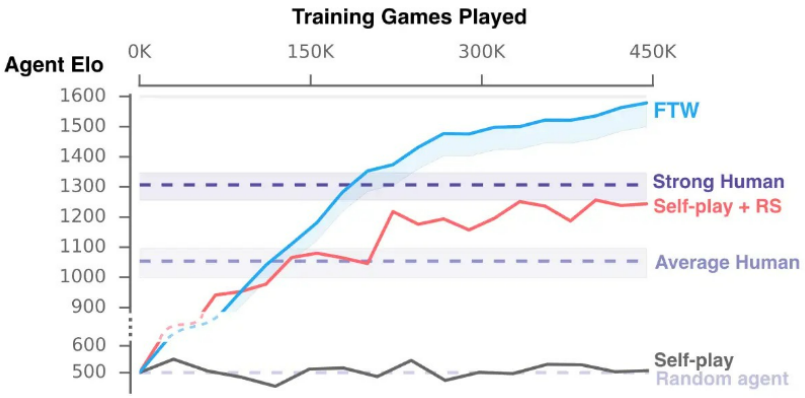


Figure 8: Capture The Flag: Confronto del valore *Elo rating* (probabilità di vincita) ottenuto dai vari sistemi.

Agli agenti non sono mai state fornite le regole del gioco: loro imparano i concetti fondamentali attraverso:

- l'attivazione di particolari neuroni che codificano gli stati più significativi del gioco (es: un compagno di squadra è in possesso della bandiera);
- l'utilizzo della memoria;
- meccanismi di *visual attention* [15].

In esperimenti successivi si è aggiunto un delay alle azioni degli agenti per renderle equiparabili ai tempi di reazione degli utenti umani: anche in questo caso gli agenti avevano performance migliori.

In conclusione, attraverso il reinforcement learning gli agenti hanno appresso automaticamente diverse behaviours (es: difesa della base, appostarsi nella base avversaria e seguire i compagni di squadra). È stato dimostrato, quindi, che tali tecniche possono essere utilizzate analogamente in altri environment (più o meno complessi), con un grado di successo molto alto.

4 Tecnologie studiate

4.1 PettingZoo

PettingZoo [14] è una libreria che fornisce un insieme di environment a cui possono interfacciarsi sistemi multi-agente; ad essa, inoltre, è associata un'API Python. L'obiettivo è quello di agevolare e accelerare la ricerca in ambito MARL, fornendo un punto di partenza per lo sviluppo di ulteriori soluzioni innovative. Di fatto, si tratta della versione multi-agente della già conosciuta **libreria *Gym*** [3], sviluppata da OpenAI e standard consolidato per il paradigma *single-agent*.

Esempio base di utilizzo di Gym

```
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for _ in range(1000):
    action = policy(observation)
    observation, reward, done, info = env.step(action)
```

Al fine di supportare un più grande numero di tipologie di environment e rendere l'API il più generale possibile per il paradigma MARL, è stato sviluppato un nuovo modello matematico di gioco, chiamato ***Agent Environment Cycle (AEC)*** (vedi sezione 4.1), completamente interscambiabile con i modelli standard già esistenti. Tra i modelli già consolidati per il MARL vi sono ***Partially Observable Stochastic Games (POSG)*** e ***Extensive Form Games (EFG)***.

Come si può vedere dall'esempio riportato di seguito, il modello POSG può essere naturalmente codificato in uno stile simil GYM dove `actions`, `observations`, `rewards`, ecc.. sono liste o dizionari di valori per gli agenti.

Esempio base di implementazione del modello POSG

```
from ray.rllib.examples.env.multi_agent import MultiAgentCartPole
env = MultiAgentCartPole()
observation = env.reset()
for _ in range(1000):
    actions = policies(agents, observation)
    observation, rewards, dones, infos = env.step(actions)
```

Questo modello matematico permette di applicare, in maniera semplice, metodi implementati per il *single-agent RL* a sistemi *multi-agent*. Tuttavia, risultano subito evidenti due problemi legati a questo tipo di soluzione:

1. supportare giochi a turni (es: scacchi) richiede il passaggio di azioni fittizie per agenti che non operano in un determinato momento;

2. modificare dinamicamente il numero di agenti (in caso di morte o creazione) è molto difficile ed, inoltre, il codice per il processo di *learning* deve far fronte a cambiamenti improvvisi nelle dimensioni delle liste.

Diversamente, il modello matematico EFG rappresenta gli scenari di gioco attraverso una struttura ad albero, dove ogni possibile sequenza di azioni viene codificata da un percorso dalla radice ad una foglia dello stesso. Un'azione verrà intrapresa al posto di un'altra in base ad un valore che definisce una distribuzione di probabilità su di esse. Successivamente ne viene riportato un esempio.

Esempio base di implementazione del modello EFG

```
import pyspiel
import numpy as np

game = pyspiel.load_game("kuhn_poker")
state = game.new_initial_state()
while not state.is_terminal():
    if state.is_chance_node():
        # Step the stochastic environment.
        action_list, prob_list = zip(*state.chance_outcomes())
        state.apply_action(np.random.choice(action_list,
        p=prob_list))
    else:
        # sample an action for the agent
        legal_actions = state.legal_actions()
        observations = state.observation_tensor()
        action = policies(state.current_agent(), legal_actions,
        observations)
        state.apply_action(action)
        rewards = state.rewards()
```

Anche in questo caso sono risultati evidenti alcuni problemi:

1. il modello e la corrispondente API sono molto complicate rispetto a POSG;
2. la definizione formale del modello include un valore di reward solo alla fine del gioco, mentre il paradigma di *reinforcement learning* richiede reward molto frequenti;
3. l'API non supporta il concetto di *continuos action*, aspetto molto importante e comune nel RL.

Dunque, *PettingZoo* è stata sviluppata seguendo i seguenti obiettivi:

1. sfruttare la **semplicità** e l'universalità di Gym, ispirando il design della nuova tecnologia ad una ampiamente conosciuta. Inoltre, si vuole supportare un'ampia raccolta di environment per attirare un grande numero di utenti;

2. deve essere una API **universale**, quindi deve essere in grado di supportare tutti i casi di utilizzo e tipi di environment (es: grande numero di agenti, agenti che muoiono e vengono creati dinamicamente, ...).

Agent Environment Cycle Games Model

In questo modello un agente, sequenzialmente, prende in input la sua *observation*, esegue *actions* e riceve *rewards* dagli altri agenti; infine, il sistema sceglie il prossimo agente che deve operare. A differenza del modello POSG, ma similmente al modello EFG, l'*AEC model* introduce un *environment agent*, il quale reagisce alle azioni degli altri agenti aggiornando se stesso.

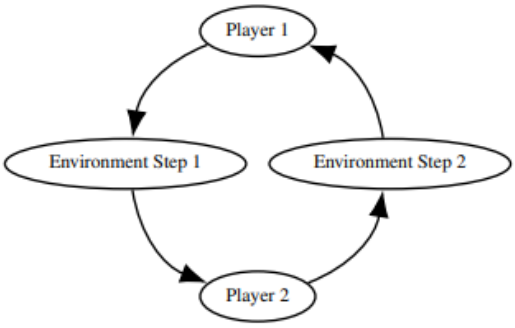


Figure 9: Diagramma del modello AEC per il gioco degli scacchi

Modellare l'environment in maniera sequenziale apporta i seguenti vantaggi:

- consente di gestire in maniera più agevole e chiara rewards da origini diverse, permettendo di implementare differenti strategie di learning;
- impedisce agli sviluppatori di inserire *race condition*;
- il modello si avvicina di più a come viene eseguito il rispettivo codice;
- consente di ottenere dei *reward* ad ogni step, come richiesto dal RL;
- è semplice abbastanza da fungere come modello mentale per i principianti;
- cambiare il numero di agenti durante l'apprendimento è meno dispendioso;
- agenti che operano sequenzialmente possono sia operare tutti simultaneamente che mettere in coda le proprie azioni, a differenza degli agenti che operano solo simultaneamente e per cui è necessario definire delle *no-op actions* (nel caso non possano agire tutti).

API

Come riportato nella sezione precedente, l'obiettivo degli sviluppatori dell'API PettingZoo era quello di sfruttare la semplicità e l'universalità di Gym, ispirando il design della nuova tecnologia ad una ampiamente conosciuta. Difatti, similmente a Gym:

- ogni agente fornisce una `action` ad una funzione `step`, la quale deve aggiornare i rewards e il turno del prossimo agente. L'agente riceverà `observation`, `reward`, `done` ed `info` come valori di ritorno;
- `render` e `close` permettono di visualizzare il frame corrente dell'environment;
- `reset`, resetta l'environment alla configurazione di partenza.

In aggiunta a quanto descritto, sono stati definiti:

- il metodo `agent_iter` che ritorna il prossimo agente che dovrà agire;
- il metodo `last` che aspetta la terminazione di tutti gli agenti prima di determinare, in maniera definitiva, il valore di ritorno dell'agente corrente. Questi valori verranno poi passati ad una `policy` per scegliere una `action`;
- l'attributo `agents` che consiste nella lista di tutti gli agenti dell'environment;
- i dizionari `rewards`, `dones`, `infos` le cui chiavi corrispondono agli agenti;
- il metodo `observation_space` ed un metodo `action_space` che ritornano rispettivamente lo spazio delle osservazioni che possono essere compiute sugli agenti e lo spazio delle azioni che possono essere compiute dagli agenti;
- il metodo `observe(agent)` che ritorna un vettore con le precedenti osservazioni dell'agente specificato;
- il metodo `state` che ritorna lo stato di un environment;
- il metodo `agent_selection` che ritorna l'agente che sta attualmente operando all'interno dell'`agent_iter`;
- l'attributo `possible_agents`, ovvero una lista di tutti gli agenti che possono esistere in un environment in un determinato punto (non valido per environment che generano un numero arbitrario di agenti);
- nel caso fosse necessario inserire un *environment agent* all'interno del sistema (vedi AEC Games Model 4.1), si dovrà generare un agente con il nome `env` e che prende `None` come azione.

Permettere l'accesso ad attributi anche di basso livello ha l'obiettivo di facilitare lo studio e la ricerca di nuove soluzioni in ambito MARL.

Esempio di utilizzo di PettingZoo

```
from pettingzoo.butterfly import pistonball_v0

env = pistonball_v0.env()
env.reset()
for agent in env.agent_iter(1000):
    env.render()
    observation, reward, done, info = env.last()
    action = policy(observation, agent)
    env.step(action)
env.close()
```

PettingZoo include 63 *environments* al suo interno ma l'implementazione di metà di questi si ferma ad un livello di ricerca, dunque non è possibile installarli mediante il comando `pip`. Un'ulteriore limitazione di quest'API viene riscontrata in termini di performance, quando il numero di agenti (o possibili agenti) supera le 10.000 unità.

Custom Environment Una feature particolarmente interessante di PettingZoo sono le API per creare un Environment personalizzato. In questo modo diventa possibile creare environment che rappresentano il proprio dominio applicativo di interesse, oltre ad utilizzare quelli presenti di default. Per potere definire un environment personalizzato è necessario reimplementare i metodi descritti nella sezione precedente.

Custom Chase Una volta studiate le precedenti API è stato sviluppato un environment di prova molto basico in cui sono presenti due agenti: un poliziotto (`Policeman`) e un ladro (`Thief`).

Gli agenti sono posti casualmente in un campo di gioco 2D, ad ogni step possono fare una mossa ciascuno e hanno due compiti opposti:

- il ladro deve scappare, quindi otterrà un punteggio positivo se alla fine dello step la sua distanza dal poliziotto sarà aumentata;
- Il poliziotto deve catturare il ladro, quindi otterrà un punteggio positivo se alla fine dello step la sua distanza dal ladro sarà diminuita;
- in caso la distanza rimanga invariata entrambi gli agenti ottengono un punteggio nullo.

Si tratta dunque di un environment di tipo *Competitive* in cui ogni agente ha cinque mosse possibili: UP, DOWN, LEFT, RIGHT e NONE.

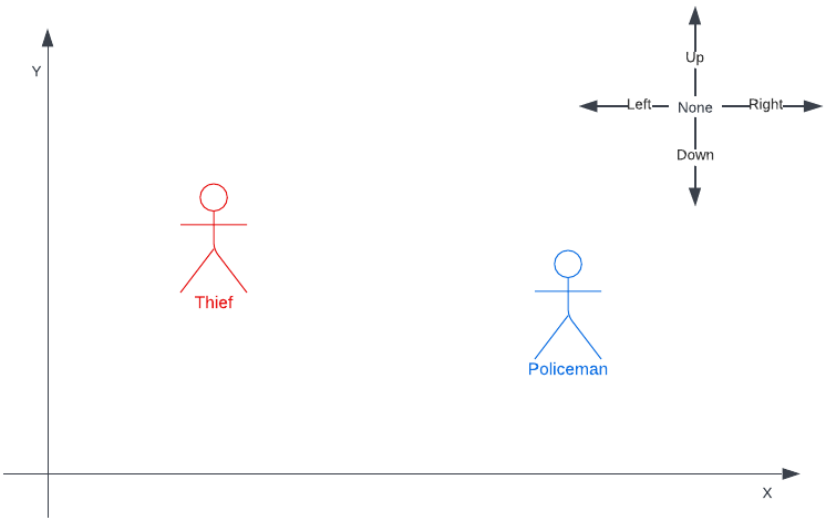


Figure 10: Custom Chase Environment

4.2 Mava

Mava [13] è un framework di ricerca sviluppato per permettere di costruire sistemi MARL in modo scalabile. Alla base di MAVA è presente ACME [8], una libreria sviluppata da DeepMind per il ***Reinforcement Learning Distribuito*** che fornisce molti algoritmi pre-implementati.

Architettura L'architettura interna di Mava è composta da diverse astrazioni: il ***System***, che rappresenta la specifica completa di un sistema MARL. Quest'ultimo, a sua volta, è composto da tre componenti:

1. ***Executor***: è un insieme di agenti. É la parte del System che interagisce con l'environment, quindi, compie un'azione per ogni agente e accumula i reward.

2. ***Trainer***: è un insieme di learner, ognuno dei quali è responsabile di interagire con il Dataset e aggiornare i parametri dei singoli agenti.

3. ***Dataset***: mantiene tutti i dati generati dall'Executor. Questo componente utilizza Reverb [4], ossia un framework sviluppato da DeepMind per gestire in maniera efficiente ed affidabile il data storage & transport.

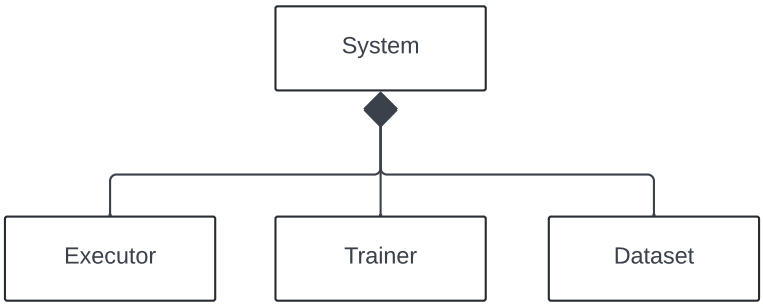


Figure 11: Componenti di MAVA

In figura [12] si può capire come i vari componenti interagiscono fra di loro in fase di learning. Nell'immagine di sinistra è presente un solo Executor quindi, sostanzialmente, si tratta di un caso non-distribuito. Nell'immagine di destra, invece, sono presenti molteplici Executor che portano avanti un learning distribuito.

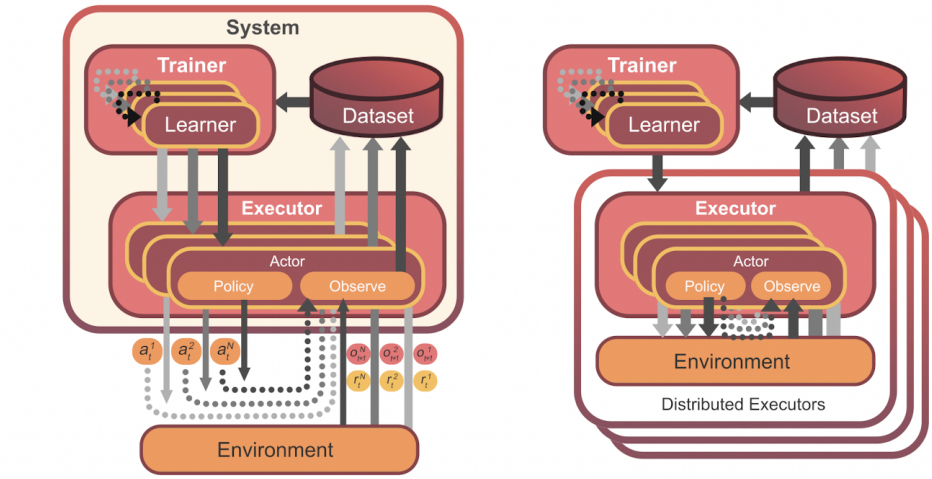


Figure 12: Interazioni dei componenti di MAVA. [13]

Interaction Loop In MAVA è stata estesa l'API proposta da DeepMind [11] per adattarla al caso dei sistemi multi-agente. Questa API è composta da:

- `Environment`, ossia un'interfaccia base per i vari environment;
- `TimeStep`, ossia un container degli output dell'environment ad ogni istante di tempo (ovvero le transizioni). In Mava è esteso come un insieme di dizionari, indicizzati con l'id dell'agente, che contengono le osservazioni, i reward e il fattore di discount;
- `specs`, che contiene le informazioni utili per descrivere le possibili azioni di un agente sull'environment (in Mava può essere diverso per ogni agente);
- `test_utils`, ossia un tool per testare che l'implementazione di un environment sia effettivamente conforme all'interfaccia `Environment`.

Esempio di Interaction Loop in Mava

```
while True:
    # make initial observation for each agent
    step = environment.reset()
    executor.observe_first(step)
    while not step.last():
        # take agent actions and step the environment
        obs = step.observation
        actions = executor.select_actions(obs)
        step = environment.step(actions)

        # make an observation for each agent
        executor.observe(actions , next_timestep=step)

        # update the executor policy networks
        executor.update()
```

Architettura del System Un altro aspetto di Mava da considerare è l'architettura del System: questa va ad impattare sul possibile flusso di informazioni fra i vari agenti. Nello specifico:

- un'architettura *decentralizzata* permette di avere solo agenti completamente indipendenti;
- un'architettura *centralizzata* permette di scambiare informazioni con un'agente "centrale" che si occupa, per esempio, di effettuare i calcoli per l'aggiornamento dei parametri;
- un'architettura *networked* permette di comunicare solo con certi agenti, quelli con cui si ha una connessione diretta (per esempio, i vicini).

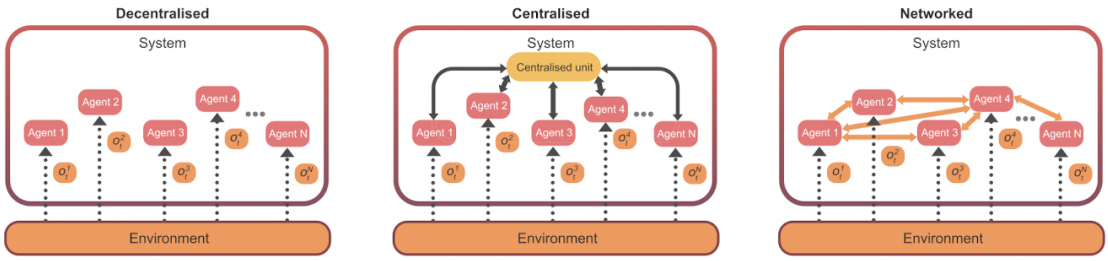


Figure 13: Architettura del System. [13]

Gestione dei componenti distribuiti La gestione dei componenti distribuiti viene effettuata sfruttando Launchpad [17]. Questo tool, sviluppato da DeepMind, consente di rappresentare i vari processi come nodi di un grafo e la comunicazione fra i vari processi come archi.

Esempio di sistema distribuito in Mava

```
#Distributed program
program = madqn.MADQN(
    environment_factory=environment_factory,
    network_factory=network_factory,
    architecture=DecentralisedPolicyActor,
    num_executors=2,
    ).build()

#Launch
launchpad.launch(
    program,
    launchpad.LaunchType.LOCAL_MULTI_PROCESSING,
)
```

L'esempio precedente mostra come può essere creato un semplice sistema MARL e la figura 14 ne mostra una rappresentazione grafica. Gli executor e il trainer vengono istanziati su tre diversi nodi; inoltre, è presente un nodo di Reverb che si occupa di gestire il dataset.

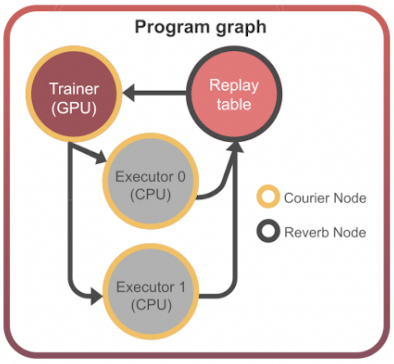


Figure 14: Esempio di grafo. [13]

5 Esperimenti

Data una definizione dei concetti principali alla base del *Multi Agent Reinforcement Learning*, si è deciso di sperimentare la combinazione di due tecnologie che riteniamo sufficientemente mature, PettingZoo [14] e Mava [13], discusse nella sezione 4 precedente. A tal fine è stato prodotto un notebook (disponibile al seguente link) che racchiude i seguenti step:

1. *Setup*, per importare i packages e i moduli necessari;
2. *Environment Selection*, per selezionare uno degli environment di default forniti dalla libreria PettingZoo;
3. *Build the system*, per definire il sistema multi-agente;
4. *Training the system*, per lanciare il training del sistema;
5. *Visualize training results using Tensorboard*, per visualizzare graficamente i valori delle metriche;
6. *View agent recordings*, per visualizzare attraverso una registrazione video le behaviour apprese dagli agenti durante il processo di learning.

Per ogni esperimento condotto è stato configurato un environment differente e i risultati ottenuti sono stati riportati nella sezione 5.2.

5.1 Metriche utilizzate

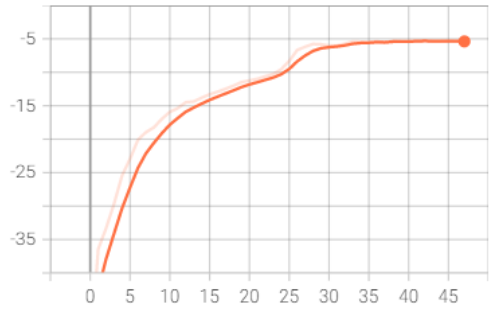
In questa sezione viene riportato un riassunto delle metriche utilizzate per valutare i risultati degli esperimenti effettuati:

- ***MeanEpisodeReturn***: questa metrica aggrega, tramite la media, i reward degli agenti in un dato episodio di learning. Questo valore può essere interpretato per capire se, collettivamente, gli agenti stanno imparando come raggiungere il goal.
- ***MeanEpisodeLength***: questa metrica rappresenta quanto gli agenti impiegano, in media, a svolgere un singolo episodio di learning. In accoppiata con la metrica precedente può fornirci informazioni importanti come, ad esempio, se gli agenti stanno imparando a risolvere un problema più velocemente mantenendo un reward adeguato, oppure se sono sia lenti che imprecisi.
- ***StdEpisodeReturn***: questa metrica rappresenta la deviazione standard dei reward ottenuti dagli agenti, in particolare è utile per capire quanto è stabile il learning.
- ***MeanStepsPerSecond***: questa metrica rappresenta il numero medio di step effettuati dagli agenti in un secondo, può essere utilizzata per capire quanto è veloce il nostro sistema ad interagire con l'environment.

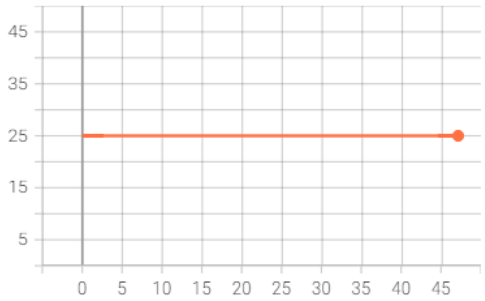
5.2 Risultati ottenuti

5.2.1 Simple

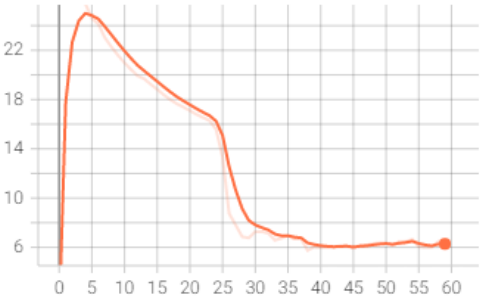
Questo environment è il più semplice fra quelli messi a disposizione da PettingZoo ed è inteso a scopo di debug: un unico agente è ricompensato in base a quanto si avvicina ad un *landmark* posizionato casualmente.



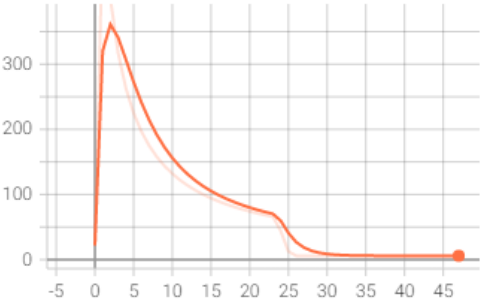
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



(d) Mean Step Per Second

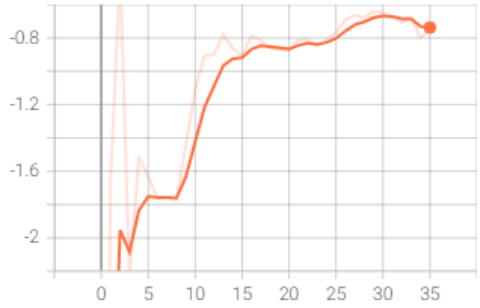
Figure 15: Simple Environment Experiment Metrics

5.2.2 Simple Adversary

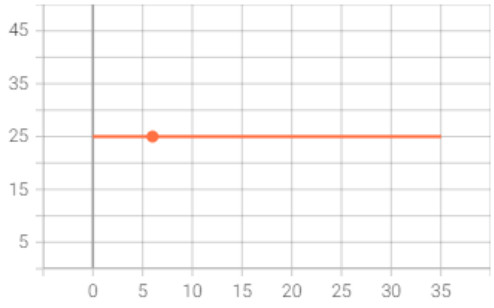
In questo environment sono presenti molteplici agenti: un agente *avversario*, N agenti *buoni* ed, inoltre, sono presenti N *landmark* (dove N di default è impostato a 2). Uno degli N landmark è detto *target* e viene usato per assegnare i reward, in particolare:

- gli agenti buoni sono ricompensati in base a quanto è prossimo il più vicino tra loro al target (gli agenti sono a conoscenza di quale sia quest'ultimo nell'insieme dei landmark);
- l'agente avversario è ricompensato in base a quanto è vicino al landmark target ma non conosce a priori qual sia degli N.

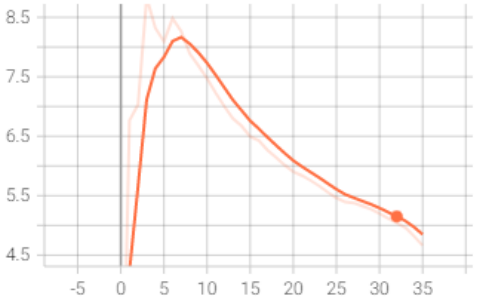
La dinamica di questi reward implica che gli agenti buoni debbano imparare a dividersi fra i vari landmark per ingannare l'avversario.



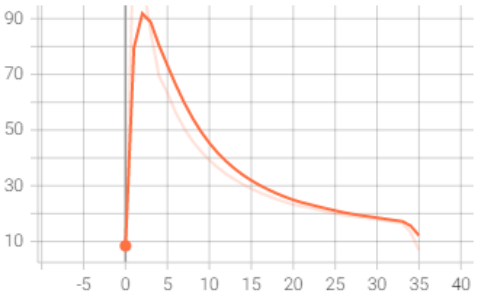
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



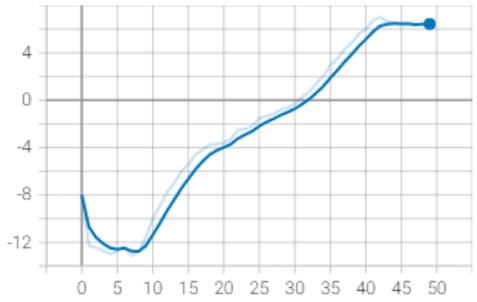
(d) Mean Step Per Second

Figure 16: Simple Adversary Environment Experiment Metrics

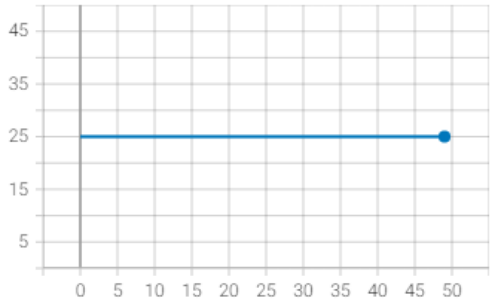
5.2.3 Simple Crypto

In questo environment sono presenti due agenti *buoni*, chiamati Alice e Bob, ed un agente *avversario*, chiamato Eve. L'obbiettivo di Alice è mandare un *messaggio* (lungo 1 bit) a Bob attraverso un *canale pubblico*. Alice e Bob hanno una *chiave di cifratura privata*, generata all'inizio di ogni episodio, che devono imparare ad usare. I reward sono assegnati nel seguente modo:

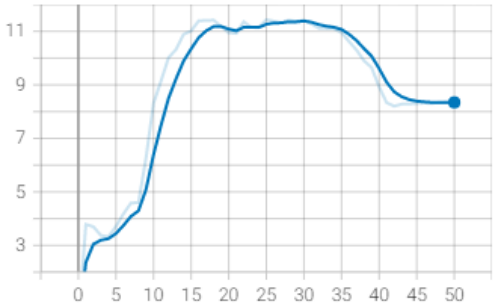
- Alice e Bob sono ricompensati con +2 se Bob riesce a decifrare il messaggio e con -2 se riesce a decifrarlo anche Eve (notare che in caso di successo di entrambi la somma è 0);
- Eve è ricompensata con +0 se riesce a decifrare il messaggio e con -2 in caso contrario.



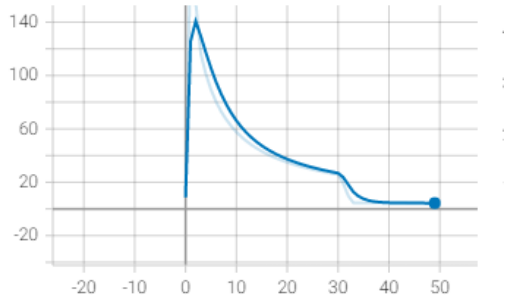
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



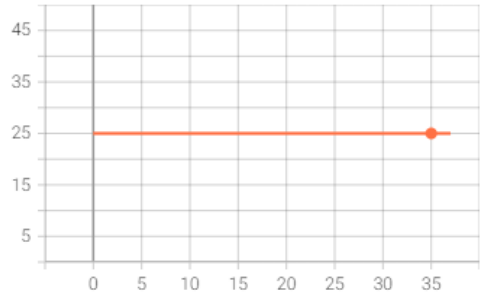
(d) Mean Step Per Second

Figure 17: Simple Crypto Environment Experiment Metrics

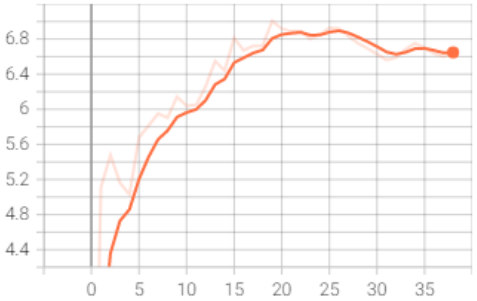
5.2.4 Simple Spread

In questo environment sono presenti N agenti ed N *landmark* (dove di default N è uguale a 3). Il goal degli agenti è avvicinarsi ai landmark evitando di collidere fra loro (reward -1 per ogni collisione fra due agenti).

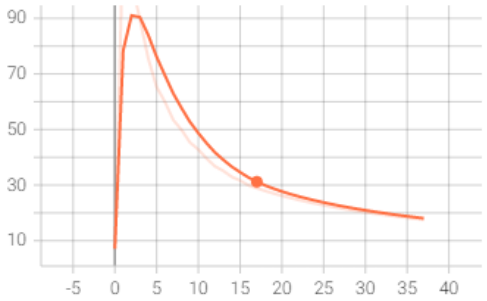
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return

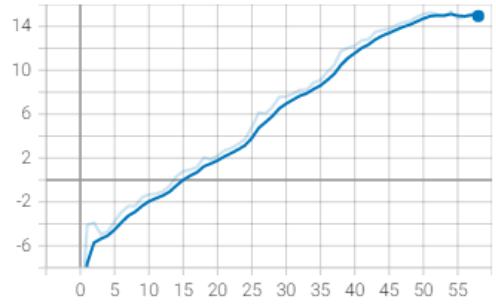


(d) Mean Step Per Second

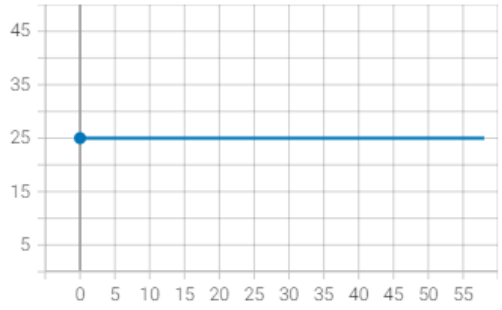
Figure 18: Simple Spread Environment Experiment Metrics

5.2.5 Simple Tag

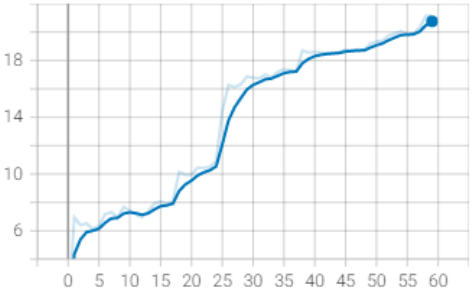
Questo è un environment del tipo *predatore-preda*. Gli agenti *preda* sono più veloci dei *predatori* e ricevono un reward negativo (-10) ogni volta che vengono colpiti da uno di essi; invece, i predatori ricevono un reward positivo (+10) ogni volta che colpiscono una preda. Inoltre, sono presenti degli *ostacoli* che limitano la fuga delle prede.



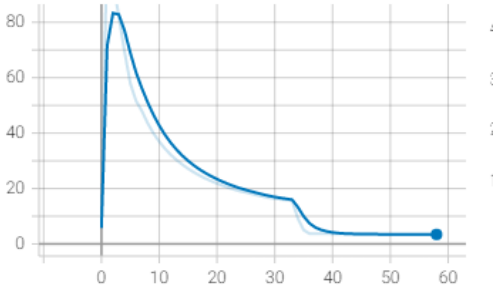
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



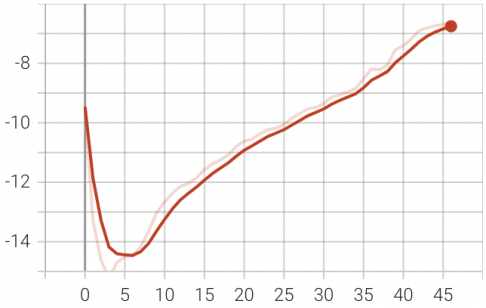
(d) Mean Step Per Second

Figure 19: Simple Tag Environment Experiment Metrics

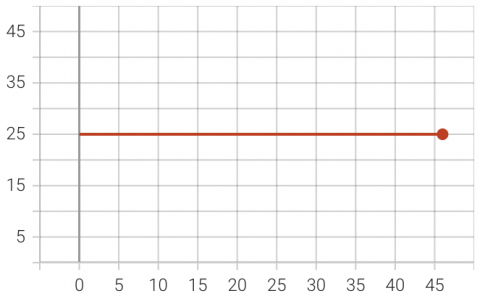
5.2.6 Simple Push

Simple Push è un environment di tipo *MPE* in cui sono presenti alcuni *landmark* e due agenti:

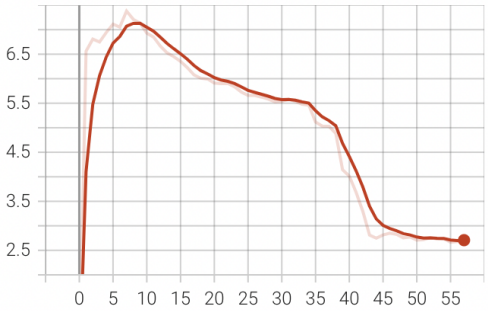
- uno degli agenti è detto *buono* ed ottiene un reward in base a quanto è vicino al landmark;
- il secondo agente è detto *avversario* ed ottiene un reward in base a due fattori:
  1. quanto è vicino al landmark;
  2. quanto riesce a far allontanare l'agente buono.



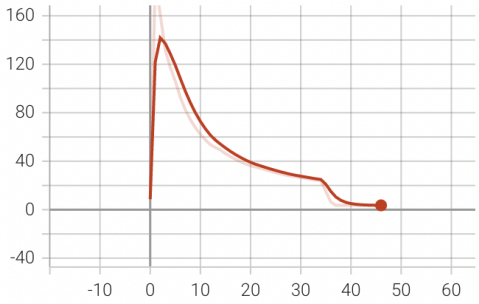
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



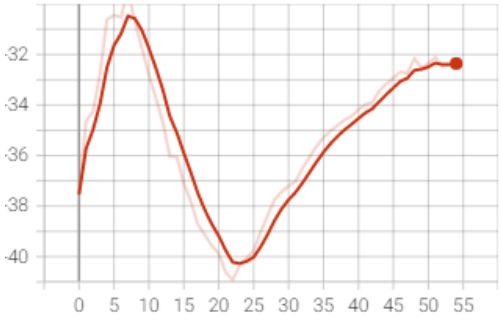
(d) Mean Step Per Second

Figure 20: Simple Push Environment Experiment Metrics

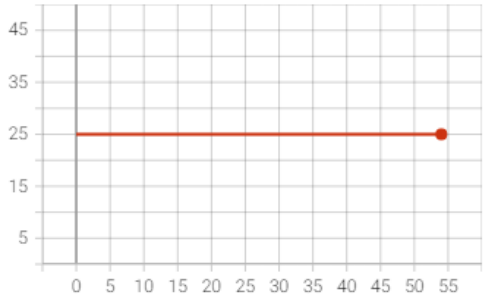
5.2.7 Simple Reference

In questo environment sono presenti due agenti e 3 *landmark*: entrambi gli agenti sono contemporaneamente *spearker* e *listener*. Ad ogni agente è assegnato un landmark *target* che, però, è conosciuto solo dall'altro agente. I reward sono assegnati in due modi:

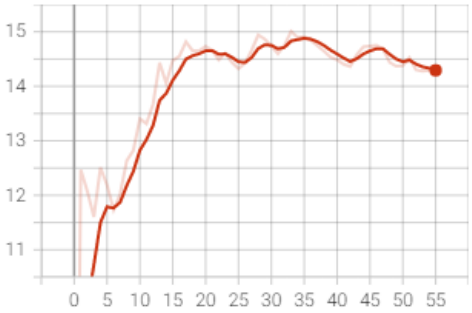
- localmente, in base alla vicinanza di un dato agente al suo landmark target;
- globalmente, in base alla distanza media di tutti gli agenti dai rispettivi landmark. In questo modo gli agenti devono imparare a cooperare per spingere l'altro ad avvicinarsi al landmark giusto.



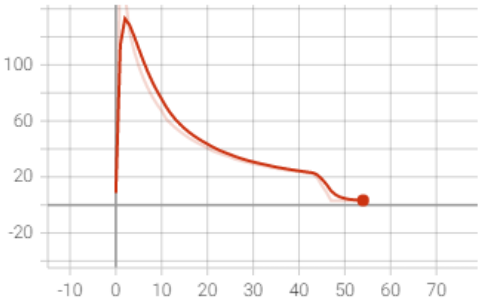
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return

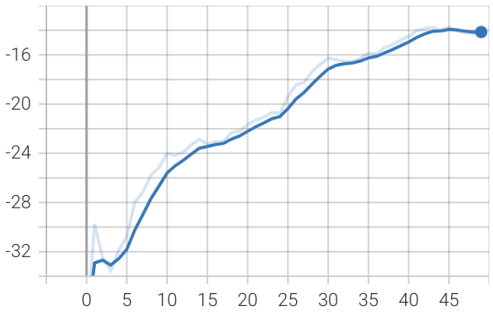


(d) Mean Step Per Second

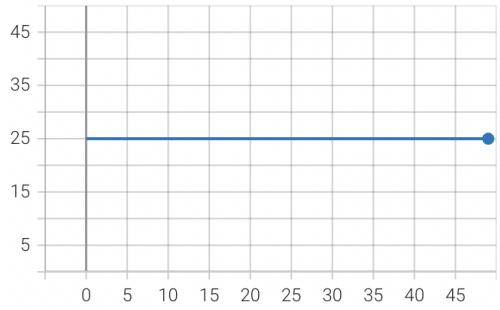
Figure 21: Simple Reference Environment Experiment Metrics

5.2.8 Simple Speaker-Listener

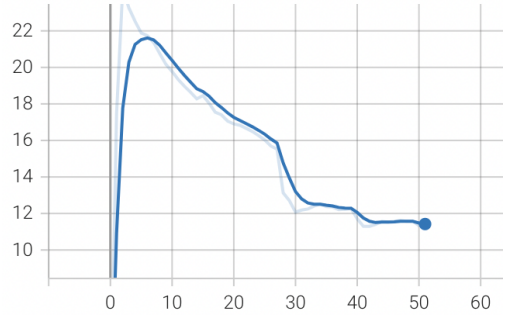
Questo environment è simile a Simple Reference visto in precedenza [5.2.7]: la differenza è che l'agente *speaker* può parlare ma non si può muovere, mentre l'agente *listener* si può muovere ma non può parlare.



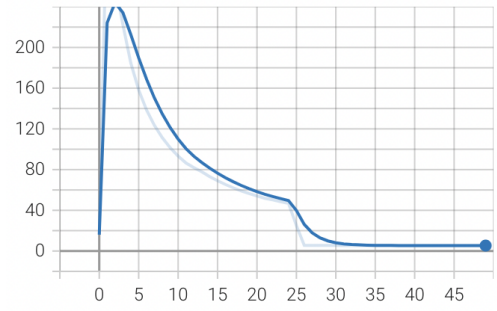
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



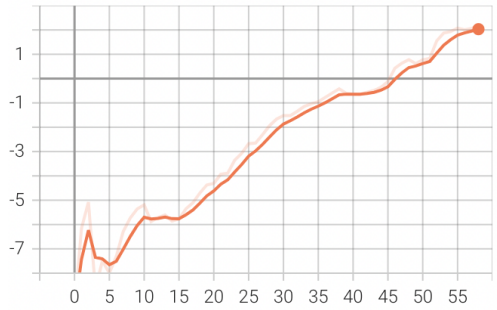
(d) Mean Step Per Second

Figure 22: Simple Speaker-Listener Environment Experiment Metrics

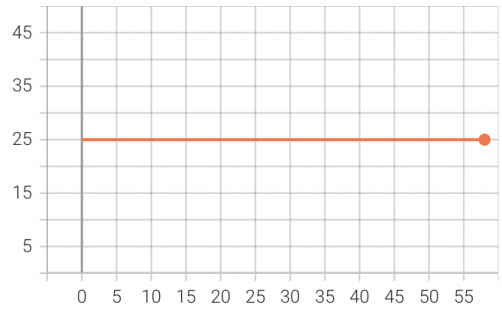
5.2.9 Simple World Comm

Questo environment è simile a Simple Tag visto in precedenza [5.2.5] ed in aggiunta:

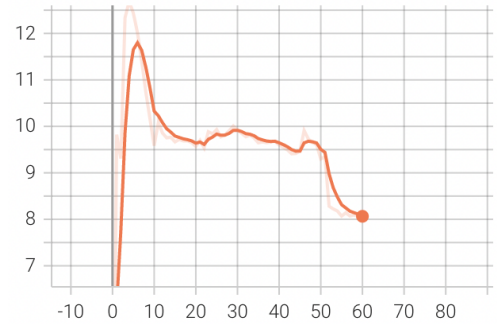
- è presente l'elemento *cibo* per cui gli agenti preda ottengono un reward positivo in base alla vicinanza ad esso;
- sono presenti delle *foreste* che limitano la visibilità di un agente;
- è presente un agente *leader* per i *predatori*, il quale vede tutte le *prede* e può coordinare gli altri predatori nell'inseguimento.



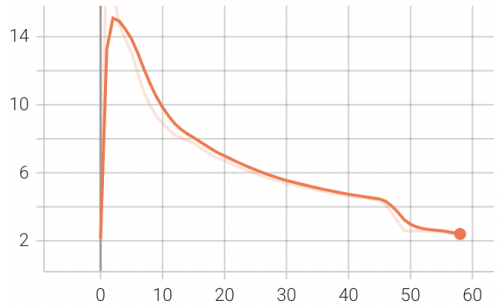
(a) Mean Episode Return



(b) Mean Episode Length



(c) Std Episode Return



(d) Mean Step Per Second

Figure 23: Simple World Comm Environment Experiment Metrics

6 Conclusioni

L'obiettivo principale di questo progetto consisteva nel studiare il dominio del *Multi-Agent Reinforcement Learning* ed alcune delle librerie più promettenti in questo ambito. A tal fine, dato che queste nozioni non sono mai state discusse in alcun corso universitario, abbiamo affrontato una prima parte di conoscenza del dominio applicativo in cui abbiamo cercato e studiato diversi articoli scientifici. Per comprendere a fondo questi sistemi, abbiamo adottato un approccio incrementale: partendo dagli approfondimenti sui sistemi ad agenti e sul Reinforcement Learning come tecnologie *stand-alone*, per poi unire queste due metodologie. Nella seconda parte abbiamo selezionato e studiato due tecnologie, cercando di capire le loro potenzialità e, soprattutto, i limiti che presentano.

La prima tecnologia studiata è stata *PettingZoo*, la quale mette a disposizione diversi environment pre-implementati oltre ad un'API per poter sviluppare un proprio *environment custom*. A posteriori, la valutazione rispetto questa libreria è risultata positiva in quanto si è rilevata molto flessibile e stabile.

Successivamente, è stato considerato il framework *Mava*, il quale permette di effettuare la fase di training degli agenti dato un environment e, nello specifico, sono stati sfruttati quelli forniti da PettingZoo. A nostro avviso questo framework ha notevoli potenzialità; tuttavia, abbiamo riscontrato anche diversi limiti. In primo luogo la documentazione è piuttosto scarsa ed, inoltre, in questi mesi la tecnologia è stata "rivoluzionata" con il passaggio da TensorFlow a Jax. Per quest'ultimo motivo sono presenti ancora molte feature non implementate, come gli environment con uno spazio di azioni continuo. Infine, non tutti gli environment di PettingZoo sono completamente funzionanti.

In conclusione, vogliamo fare una riflessione riguardo questi sistemi: le tecniche come il Reinforcement Learning offrono notevoli potenzialità ma, guardando l'altro lato della medaglia, sono ancora utilizzate come *black-box* data l'immaturità delle tecniche di *explainability* ed *interpretability*. Uno sviluppo interessante potrebbe vertere verso un mix di tecniche di machine learning con modelli come il *BDI (beliefs, desires, intentions)*.

References

[1] Gianluca Aguzzi. *Collective Learning*. Lesson on Collective Learning. 2021.

[2] Bowen Baker et al. *Emergent Tool Use From Multi-Agent Autocurricula*. 2019. DOI: 10.48550/ARXIV.1909.07528. URL: https://arxiv.org/abs/1909.07528.

[3] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[4] Albin Cassirer et al. *Reverb: A Framework For Experience Replay*. 2021. arXiv: 2102.04736 [cs.LG].

[5] Caroline Claus and Craig Boutilier. "The Dynamics of Reinforcement Learning in Cooperative Multiagent Systems". In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. Madison, Wisconsin, USA: American Association for Artificial Intelligence, 1998, pp. 746–752. ISBN: 0262510987.

[6] Sven Gronauer and Klaus Diepold. "Multi-agent deep reinforcement learning: a survey". In: *Artificial Intelligence Review* 55.2 (2022), pp. 895–943. DOI: 10.1007/s10462-021-09996-w. URL: https://doi.org/10.1007/s10462-021-09996-w.

[7] Pablo Hernandez-Leal et al. *A Survey of Learning in Multiagent Environments: Dealing with Non-Stationarity*. 2017. DOI: 10.48550/ARXIV.1707.09183. URL: https://arxiv.org/abs/1707.09183.

[8] Matthew W. Hoffman et al. *Acme: A Research Framework for Distributed Reinforcement Learning*. 2020. DOI: 10.48550/ARXIV.2006.00979. URL: https://arxiv.org/abs/2006.00979.

[9] Max Jaderberg et al. "Human-level performance in 3D multiplayer games with population-based reinforcement learning". In: *Science* 364.6443 (2019), pp. 859–865. DOI: 10.1126/science.aau6249. eprint: https://www.science.org/doi/pdf/10.1126/science.aau6249. URL: https://www.science.org/doi/abs/10.1126/science.aau6249.

[10] Michael L. Littman. "Markov games as a framework for multi-agent reinforcement learning". In: *Machine Learning Proceedings 1994*. Ed. by William W. Cohen and Haym Hirsh. San Francisco (CA): Morgan Kaufmann, 1994, pp. 157–163. ISBN: 978-1-55860-335-6. DOI: https://doi.org/10.1016/B978-1-55860-335-6.50027-1. URL: https://www.sciencedirect.com/science/article/pii/B9781558603356500271.

[11] Alistair Muldal et al. *dm_env: A Python interface for reinforcement learning environments*. 2019. URL: http://github.com/deepmind/dm_env.

[12] James Odell. "Objects and agents compared". In: *Journal of object technology* 1.1 (2002), pp. 41–53.

[13] Arnu Pretorius et al. "Mava: A Research Framework for Distributed Multi-Agent Reinforcement Learning". In: *arXiv preprint arXiv:2107.01460* (2021). URL: https://arxiv.org/pdf/2107.01460.pdf.

[14] Justin K. Terry et al. "PettingZoo: Gym for Multi-Agent Reinforcement Learning". In: *CoRR* abs/2009.14471 (2020). arXiv: 2009.14471. URL: https://arxiv.org/abs/2009.14471.

[15] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[16] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[17] Fan Yang et al. "Launchpad: A Programming Model for Distributed Machine Learning Research". In: *arXiv preprint arXiv:2106.04516* (2021). URL: https://arxiv.org/abs/2106.04516.