**TESI DI LAUREA in Ingegneria Informatica, percorso ICT Engineering, Business and Innovation**

**MULTI AGENT LEARNING WITH LEGO® SPIKE PRIME EDUCATION**

**Index**

1. Abstract (Italian and English)
2. Introduction
3. Multi Agent Reinforcement Learning (State of the Art)
   1. Reinforcement Learning, what it is.
   2. Multi Agent Reinforcement Learning
   3. Multi Agent RL Applications
   4. Multi Agent RL Limits
4. Formal Model Definition
   1. Definition of the Markov Game for multi agent learning.
   2. Definition of an MDP.
   3. Demonstration that in centralized settings, two MG converge in a MDP.
5. The Framework Used
   1. LEGO® Spike Prime Education.
   2. Explanation of the hardware and its possibilities.
   3. Beyond Modular Programming and Pybricksdev Framework.
   4. Main action used on the robot.
6. Research and Experimentation
   1. Grid-World definition
   2. Multi-Agent Q-Learning, results and observations.
   3. Centralized Q-Learning, results and observations.
   4. Multi-Agent FAQ-Learning, results and observations.
7. Conclusion

**Multi Agent Reinforcement Learning (State of the Art)**

**Reinforcement Learning**

Al fine di comprendere al meglio gli argomenti trattati all’interno di questa tesi, è doveroso presentare il punto cardine intorno al quale ruota la sperimentazione e le successive conclusioni che verranno presentate all’interno di questo documento, l’apprendimento basato su Reinforcement Learning.

Allo Stato dell’Arte, il Reinforcement Learning è un sottotipo di Unsupervised Learning nell’ambito dell’apprendimento automatico in cui un’agente impara a compiere azioni all’interno di un ambiente ignoto al fine di massimizzare una funzione di ricompensa. Esso appartiene al ramo del Machine Learning chiamato Unsupervised Learning proprio perché gli input del modello sono ignoti e vengono appresi con “l’esperienza’ stessa dell’agente all’interno dell’ambiente.

Al fine di definire al meglio il Reinforcement Learning, è necessario introdurre diversi elementi che lo compongono:

* **Agente**: l'entità che apprende e prende decisioni. Può essere un programma, un robot o qualsiasi sistema in grado di interagire con un ambiente.
* **Ambiente**: il contesto in cui l'agente opera. Può essere completamente o parzialmente osservabile dall’agente.
* **Azioni**: il set di mosse che un agente può prendere una volta osservato lo stato corrente dell’ambiente.
* **Stato**: la rappresentazione dell'ambiente in un dato momento. Lo stato è ciò che l'agente osserva per prendere decisioni.
* **Feedback o Reward**: un valore numerico che l'agente riceve dopo aver compiuto un'azione in uno stato specifico. L'obiettivo dell'agente è massimizzare la somma cumulativa delle ricompense nel tempo.

Il processo di apprendimento nel Reinforcement Learning coinvolge l'agente che esplora l'ambiente, apprende quali azioni sono più vantaggiose in determinati stati e aggiorna la propria strategia di decisione per massimizzare la ricompensa accumulata.

**Multi Agent Reinforcement Learning (MARL)**

Ma, come si evince dal titolo stesso di questo elaborato, il nostro focus verterà su una branca specifica del Reinforcement Learning, ovvero il Multi Agent Reinforcement Learning.

Per capire al meglio le impostazioni di questo tipo di problema d’apprendimento, è necessario introdurre i sistemi multi-agente.

Un sistema multi-agente è definito come un gruppo di entità autonome che interagiscono tra loro e condividono lo stesso ambiente, che usualmente percepiscono attraverso sensori e in cui si muovono attraverso attuatori, ma non solo.

Sebbene un agente all’interno di questo tipo di sistemi possa essere programmato per comportarsi in un determinato modo in precedenza, risulta comunque necessario far si che lui apprenda parte del suo comportamento “online”, in modo tale da migliorare le singole prestazioni, ma anche quelle di tutto l’ambiente. Ciò avviene solitamente perché la complessità dell’ambiente rende difficile, o addirittura impossibile, la progettazione a priori di una policy ottima. *( L. Busoniu, R. Babuska and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," in IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 38, no. 2, pp. 156-172, March 2008, doi: 10.1109/TSMCC.2007.913919*)

Al fine di garantire questo apprendimento “online”, è necessario appunto ricorrere a specifici algoritmi che si rifanno ai concetti di esplorazione e valorizzazione del sopra citato Reinforcement Learning, introducendo quindi il Multi-Agent Reinforcement Learning (MARL).

La presenza di agenti multipli modifica di gran lunga le possibilità di apprendimento poiché ogni singolo agente è costretto ad “imparare” considerando anche altri agenti che si muovono nel suo stesso ambiente.

In un ambiente MARL gli agenti interagiscono tra di loro in diversi modi: essi devono imparare a prendere decisioni in modo cooperativo o competitivo a seconda dell’obiettivo prefissato e massimizzare, appunto, le loro ricompense complessive.

È per questo che, in aggiunta ai sopra citati elementi che compongono un algoritmo di Reinforcement Learning, vanno menzionati dei nuovi che risultano necessari in questo nuovo scenario:

* **Agenti Multipli**: ci sono più agenti che coesistono e interagiscono nell'ambiente. Ogni agente può avere obiettivi individuali, e l'apprendimento avviene considerando sia le azioni dell'agente stesso che quelle degli altri agenti.
* **Comunicazione e Coordinamento**: gli agenti possono comunicare tra di loro per scambiare informazioni, coordinarsi o prendere decisioni condivise. Questo aspetto è cruciale in scenari in cui il successo di un agente dipende dalle azioni degli altri.
* **Competizione o Collaborazione**: A seconda del contesto, gli agenti possono essere in competizione tra di loro per raggiungere obiettivi individuali o possono collaborare per massimizzare un obiettivo comune.
* **Problemi di Non-Stationarity**: L'ambiente è spesso non stazionario a causa delle azioni in continua evoluzione degli altri agenti, il che rende la pianificazione e l'apprendimento più complessi rispetto al caso in cui un singolo agente agisce da solo.

Questi elementi tracciano anche quelle che sono le sostanziali differenze tra un Single Agent RL e un Multi Agent RL. Difatti, se il primo può essere formalmente ricondotto ad un Processo Decisionale Markoviano (MDP) dove appunto le decisioni dell’agente in ogni singolo stato sono dettate da singole probabilità dipendenti solo dallo stato osservato; il secondo perde questa proprietà decisionale essendo influenzato anche dai comportamenti degli altri agenti, evidenziando, appunto, la non stazionarietà dei singoli stati dell’ambiente e diventando formalmente non più un MDP, bensì un Gioco Stocastico o Gioco Markoviano che trova le sua fondamenta nella teoria dei giochi. (*L. Busoniu, R. Babuska and B. De Schutter, "A Comprehensive Survey of Multiagent Reinforcement Learning," in IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol. 38, no. 2, pp. 156-172, March 2008, doi: 10.1109/TSMCC.2007.913919*)

Analizzeremo meglio questi formalismi nel capitolo successivo ad essi dedicato.

**Examples and Applications**

I sistemi multi-agente stanno trovando applicazioni in un’ampia varietà di settori, tra cui robotica, controllo distribuito, gestione delle risorse, sistemi di supporto decisionale collaborativo, data mining, ecc.

In questa sezione ne prenderemo alcuni come esempio proprio per comprendere al meglio quelle che sono le possibilità che algoritmi MARL possono offrire e in quali diversi ambiti possono essere applicati. Non useremo questo approfondimento per scendere nel dettaglio delle diverse implementazioni, bensì per avere un quadro generale su cui basare le nostre successive analisi.

**Google® DeepMind**

**Google DeepMind** è un primo esempio di azienda, acquistata da Google nel 2015, che ha fatto del machine learning e dell’intelligenza artificiale i suoi punti cardine. Il loro obiettivo dichiarato è proprio quello di “ Risolvere l’intelligenza e quindi utilizzarla per affrontare importanti sfide della società". ([*https://deepmind.google/*](https://deepmind.google/)).

I loro progetti legati all’intelligenza artificiale sono svariati e tutti utilizzano diversi tipi di algoritmi.

Un esempio di utilizzo di MARL è “AlphaStar”, un software in grado di giocare attivamente al videogioco StarCraft II che ha conseguito nell’Agosto del 2019, a 7 mesi dal suo lancio, il titolo di Grandmaster all’interno del videogame. All’interno del software, come scritto nella documentazione presente sul sito, sono presenti tecniche di apprendimento automatico generiche, tra cui reti neurali, self-play via Reinforcement Learning, Multi-Agent e Imitation Learning, per apprendere direttamente dai dati di gioco con tecniche general pourpose. *(“AlphaStar: Grandmaster level in StarCraft II using multi-agent reinforcement learning”, The Alphastar Team, 30 October 2019*). Le tecniche Multi-Agent risultano quindi utili al software durante l’apprendimento di policy ottime al fine di superare e sconfiggere le policy di altri agenti presenti nel videogioco. AlphaStar è un primo esempio di Adversarial Multi-Agent Learning, accompagnato però da ulteriori metodi d’apprendimento per irrobustirne l’efficacia. Insieme ad AlphaStar, altri progetti di DeepMind utilizzano MARL per i loro scopi, come AlphaGo (gioco del Go) e AlphaZero (gioco degli Scacchi).

**Drones and Field Coverage**

In March 2018, some researchers from the Cornell University proposed a cooperative and distributed Multi Agent Reinforcement Learning of drones for field coverage.

The target where Unmanned Aerial Vehicles (UAV) that needs to explore the area in the most efficient way. The proposed MARL algorithm allows UAVs to learn cooperatively to provide a full coverage of an unknown field of interest while minimizing the overlapping sections among their field of views. Here the problem was solved using a Multi-Agent Q-Learning approach together with a game-theoretical correlated equilibrium mechanism and a function approximation used to face the joint-action selection problem and the high dimensions of the state-actions field.

(Huy Xuan Pham, Hung Manh La, David Feil-Seifer, Aria Nefian, “Cooperative and Distributed Reinforcement Learning of Drones for Field Coverage”, 20 March 2018)

**Fleet Management**

In 2018, a multi-agent reinforcement learning framework was proposed to manage fleet of online ride-sharing platforms such as Uber and Lyft. The authors introduced two algorithms to allocate cars (represented as agents) to specific zones in a city divided into hexagonal grids. Computational efficiency was enhanced using geographical and collaborative contexts, reducing action spaces and avoiding conflicting movements among agents.

They promoted this solution as a way to make efficient the fleet management strategy, improving the utilization of transportation resources and also increasing the revenue and the customer satisfaction. (K Lin, R Zhao, Z Xu, J Zhou, "Efficient large-scale fleet management via multi-agent deep reinforcement learning“, 2018)

**Lithium-Ion Battery Scheduling**

Sui and Song in 2020 proposed a reinforcement learning framework for solving battery scheduling problems in order to extend the lifetime of batteries used in electrical vehicles (EVs), cellular phones, and embedded systems. The Multi-Agent perspective was obtained by considering each battery as a single agent, generating a cooperative MARL problem that optimizes the charge and the battery’s thermal profile.

This because battery pack lifetime has often been the limiting factor in many of today’s smart systems and smart charge-discharge scheduling of battery packs is essential to obtain super linear gain of overall system lifetime. (Y Sui, S Song , “A multi-agent reinforcement learning framework for lithium-ion battery scheduling problems”)

Queste erano solo alcune delle svariate applicazioni di questo tipo di algoritmi. Esse dimostrano come i campi che necessitano di miglioramenti sotto il punto di vista del Multi-Agent Learning sono molto ampi e variegati. Ciononostante, questo tipo di implementazioni presentano diversi limiti che andremo prontamente ad analizzare.

**MARL Limitations**

Una volta comprese le impostazioni e il funzionamento degli algoritmi MARL, insieme ad alcune loro possibili applicazioni, è anche possibile intravedere quelle che sono evidenti limitazioni legate direttamente a quelli che sono gli elementi aggiuntivi, rispetto ad algoritmi Single Agent, che li connotano.

La peculiarità di sistemi Single Agent è quella di poter essere ricondotti a dei Processi Decisionali Markoviani, per i quali sussiste l’assunzione markoviana che ci garantisce che l’apprendimento è strettamente legato all’ultimo stato osservato, ovvero quello presente. Questo elimina completamente l’influenza degli stati passati durante il miglioramento dei parametri d’apprendimento, tipico appunto di sistemi stazionari.

Ad esempio, all’interno di algoritmi come il Q-Learning, il più diffuso nell’apprendimento single-agent, la regola di update prescinde esclusivamente dallo stato corrente e dallo stato futuro, senza fare alcun riferimento a quelli attraversati in precedenza.

Negli algoritmi Multi-Agent la **stazionarietà** cessa d’esistere con l’introduzione di più agenti che influiscono attivamente sull’apprendimento dei restanti. Ad esempio, una policy che risulta ottimale in un certo istante per un singolo agente potrebbe non esserlo nell’istante successivo a causa di un secondo agente che influenza “negativamente” l’ambiente e gli stati ottimali per il primo, rendendoli, per esempio, inutilizzabili.

Questo elimina l’assunzione Markoviana e aumenta la complessità e la gestione dell’apprendimento, introducendo nuove sfide e nuove tematiche d’interesse al fine di ottimizzare e assicurare il processo d’apprendimento.

L’aumento del numero di agenti incide anche su altre caratteristiche d’interesse di questi algoritmi, come la **scalabilità**. L’aumento del numero d’agenti all’interno dell’ambiente non solo aumenta la complessità di gestione degli stessi, ma anche la complessità dell’ambiente stesso, facendo incrementare esponenzialmente lo spazio congiunto delle azioni possibili (dovendo considerare insieme le azioni di un numero sempre più grande di agenti) portando le risorse computazionali necessarie a crescere esponenzialmente insieme al numero di agenti.

L’ **osservabilità parziale** è un altro dei limiti legati agli ambienti MARL. La maggior parte degli algoritmi esistenti hanno come assunzione proprio la completa osservabilità di uno stato, cosa che però in esempi pratici è difficile da replicare per tutti gli agenti che potrebbero essere più o meno consapevoli del loro stato e di quello degli altri. Ad esempio, nei casi reali gli agenti possono essere provvisti di diversi sensori o essere limitati sotto quel punto di vista, rendendo la completa osservabilità dello stato impossibile.

Il problema dell’osservabilità parziale può essere risolto introducendo la comunicazione tra i diversi agenti, che diventano quindi in grado di aggiornarsi a vicenda sullo stato attuale del sistema. Questo, però, introduce nuove limitazioni legate proprio alla comunicazione tra agenti, i quali devono poter comunicare fra loro in maniera costruttiva senza cadere in situazioni di comunicazione superflua o fuorviante. (Canese L, Cardarilli GC, Di Nunzio L, Fazzolari R, Giardino D, Re M, Spanò S. Multi-Agent Reinforcement Learning: A Review of Challenges and Applications. Applied Sciences. 2021; 11(11):4948.)

Inoltre, gli algoritmi MARL presentano problematiche anche nel bilanciamento tra cooperazione e competizione, che fanno direttamente capo ad una scelta adeguata di incentivi e ricompense. Difatti, comprendere quali ricompense sono ottimali e che tipo di comportamento favoriscono tra gli agenti può richiedere analisi approfondite e robuste analisi di sensibilità sull’algoritmo. Sul lato comunicativo, è importante comprendere se un approccio centralizzato, dove un master controlla e monitora tutte le azioni e gli stati degli agenti, oppure un approccio decentralizzato dove ogni agente resta consapevole soltanto del proprio stato e delle proprie azioni, risultino i migliori e più efficienti.

Se nel primo caso, un ambiente centralizzato garantisce una completa osservabilità dell’ambiente d’apprendimento, il rischio è quello di dover gestire ambienti troppo complessi che richiedano ingenti risorse computazionali, sfociando così nella famosa “Curse of Dimensionality”; nel secondo caso, invece, l’assenza completa o parziale di comunicazione tra le due parti potrebbe risultare in un esplorazione inefficacie e inconcludente, richiedendo quindi accorgimenti sul trovare sub-ottimi di esplorazione e equilibri virtuali tra i vari agenti.

È perciò evidente come non ci sia una vera e propria standardizzazione su questo tipo di algoritmi, rendendo ogni implementazione fine a sé stessa, con la necessità di doverla personalizzare a seconda del contesto e degli agenti che andremo ad utilizzare; questo problema rende difficile la serializzazione, la replica e soprattutto la comparazione di diversi algoritmi Multi-Agent.

I limiti degli algoritmi MARL sono evidenti e complessi, ma non invalicabili. Quello che proponiamo noi attraverso questo studio di tesi è proprio quello di trovare, attraverso prove ed esperimenti, il giusto compromesso che dimostra come un algoritmo Multi-Agent possa effettivamente funzionare in determinate condizioni, monitorandone e valutandone le prestazioni, paragonandola a suoi simili per trarne le giuste conclusioni. L’obiettivo è comprendere quali di queste precedenti limitazioni possono essere superate e quali no. Prima di farlo, dobbiamo definire al meglio quelli che sono i modelli su cui si è basata la nostra ricerca e perché li abbiamo usati.

**The Agents’ Framework**

Una volta definiti i modelli su cui si baserà la nostra sperimentazione, vogliamo presentare quelli che saranno i nostri soggetti di studio, ovvero i singoli agenti che prenderanno parte all’apprendimento MARL e che si muoveranno negli ambienti sopra definiti.

Al fine di avere una sperimentazione snella e semplice, non abbiamo optato per agenti complessi quali possono essere robot da laboratorio sofisticati, con diversi tipi di attuatori e gradi di libertà, i quali avrebbero aggiunto complessità superflua al nostro lavoro e soprattutto non necessaria. È per questo che la nostra scelta è ricaduta su un modello di robot che ha come primo obiettivo quello di avvicinare ragazzi delle scuole medie al mondo della programmazione e della logica: il robot LEGO® Education Spike Prime.

Il suo utilizzo rappresenta la doppia sfida di verificare l’efficienza di algoritmi MARL e farlo su hardware che non sono stati pensati per questo tipo di lavori.

Successivamente introdurremo il robot, le sue caratteristiche e il suo funzionamento, cercando di spiegare come è stato possibile utilizzarlo nella nostra ricerca.

**LEGO® Education Spike Prime**

LEGO® Education Spike Prime è un set LEGO® formato da un set base e una sua espansione che permette di ottenere il robot completo che potete vedere in figuraX. Il set complessivo è formato da un totale di 1132 pezzi che possono essere usati per creare la “Struttura Motrice Avanzata” o altri robot più semplici che, insieme ad oggetti di scena, possono completare le sfide proposte da LEGO® stessa.

Il nostro focus però sarà sulla “Struttura Motrice Avanzata”, che per comodità chiameremo Spike, e sulle sue componenti, quella che poi è stata effettivamente usata durante la sperimentazione.

Spike è un robot che presenta due ruote motrici da 8.8 cm e due ruote libere che lo aiutano nei movimenti di curvatura e virata. Oltre alla struttura in mattoncini LEGO® che forma il telaio del veicolo, Spike presenta un variegato set di sensori e motori:

* Sensore di Distanza: incorpora “occhi” programmabili a LED e misura ad ultrasuoni distanze comprese tra 1 e 200 cm, con una precisione di ±1 cm;
* Sensore di Forza: misura forze con valori massimi di 10 Newton (~1 kg). Può fungere anche da sensore di contatto poiché in grado di rilevare anche lievi pressioni;
* 2x Sensori di Colore: è in grado di distinguere ben 8 colori con una frequenza di campionamento a 1 kHz e misurare la luce riflessa sul suo “occhio” e la luce ambientale;
* 2x Motori Angolari Medi: presentano un sensore di rotazione integrato con posizionamento assoluto e precisione fino ad 1 grado;
* 2x Motori Angolari Grandi: più stabili rispetto a quelli medi e perfetti per eseguire controlli in linea retta.

Tutti gli elementi sopraindicati presentano degli adattatori a 6 pin per essere collegati ad altri dispositivi. In particolare, possono essere collegati all’elemento centrale del sistema, che rappresenta il vero e proprio hardware di Spike, l’Hub Intelligente. Esso ha un peso di 63 g (senza batteria) e delle dimensioni di L88.0 x W56.0 x H32.0 mm, ed è caratterizzato da:

* 6 porte input/output utilizzate per collegare sensori e motori all’hub;
* 4 pulsanti, 1 pulsante centrale di avvio, 2 pulsanti direzionali e un pulsante per attivare il pairing Bluetooth;
* Connettività Bluetooth Low Energy (BLE);
* Una matrice colorata personalizzabile di dimensioni 5x5 che può anche fungere da display;
* Un giroscopio a 6 assi in grado di monitorare la posizione relativa dell’hub (3 assi dedicati al giroscopio e 3 assi dedicati all’accelerometro);
* A rechargeable lithium-ion battery da 2100 mAh;
* Uno speaker;
* Un sistema operativo MicroPython programmabile montato su un hardware che presenta:
  + Un processore a 100MHz M4 320 KB RAM 1M FLASH;
  + 32 MB di memoria per programmi, suoni e altri file;

Tutte le caratteristiche tecniche sono state verificate su <https://assets.education.lego.com/>.

**Programmabilità dell’Hub**

Il sistema operativo in MicroPython supporta la lettura ed esecuzione di programmi scritti in due diversi tipi di linguaggi.

Come linguaggio principale abbiamo Python, con il supporto della libreria dedicata a tutte le azioni di Spike. Sebbene la libreria comprenda una vasta gamma di funzioni atte a controllare diversi aspetti del robot, passando dai motori fino ad arrivare a tutti i sensori, l’ambiente in MicroPython risulta al quanto chiuso e risulta difficile integrare librerie esterne quali NumPy o Matplotlib, utili per il tipo di algoritmi che andremo ad implementare.

Ancora più circoscritto è il linguaggio di coding “drag and drop” basato su Scratch, pensato appositamente per avvicinare i ragazzi al mondo della programmazione.

Ai fine della nostra sperimentazione, tuttavia, è stato necessario far interagire i robot con un computer esterno per alleviare il peso computazionale di determinati algoritmi sull’hub stesso. Per poter effettuare questo tipo di collegamento ci siamo serviti in primis della connettività Bluetooth di Spike e poi successivamente di due librerie Python di terze parti per la connettività e il controllo dei motori e della sensoristica.

L’utilizzo di un firmware differente da quello LEGO®, sempre basato su un sistema operativo MicroPython, ci ha permesso di spostare gran parte del codice su computer, lasciando al robot solo l’elaborazione delle azioni ricevute; ma questi aspetti verranno meglio approfonditi successivamente.

**Connettività dell’Hub**

Al fine di poter sfruttare la connessione Bluetooth presente sull’Hub e interagire con un computer utilizzato come Client in una comunicazione Peer-to-Peer è stato necessario servirsi della libreria Python chiamata Bleak.

Per comunicare con Spike, la libreria Bleak si serve di un preciso layer di comunicazione del protocollo BLE chiamato Generic Attribute Profile (GATT). Esso è perfetto per essere utilizzato in un’applicazione lato Client per far interagire due device connessi.

Dal punto di vista del layer GATT, i dati sono passati e salvati sottoforma di caratteristiche al suo interno. Il device che contiene il database delle caratteristiche che possono essere lette o scritte è considerato un GATT Server; un GATT Client, invece, è colui che scrive su queste determinate caratteristiche. Nel nostro specifico caso, il layer GATT è individuato da:

* Un servizio contenente le caratteristiche GATT avente UUID = "6E400001-B5A3-F393-E0A9-E50E24DCCA9E";
* Una caratteristica in lettura avente UUID = "6E400002-B5A3-F393-E0A9-E50E24DCCA9E";
* Una caratteristica in scrittura avente UUID = "6E400003-B5A3-F393-E0A9-E50E24DCCA9E".

Per poter gestire al meglio la comunicazione è la connessione robot – dispositivo, è stato necessario creare un oggetto Python denominato “Connection” che attraverso l’utilizzo della libreria Bleak è in grado di gestire le connessioni lato client (il computer).

Un’ istanza di questo oggetto viene creata avente i seguenti attributi:

* hub\_name = esso rappresenta una stringa contenente il nome con cui il robot da connettere al client si identifica nella rete BLE;
* service = lo “UART SERVICE UUID” del servizio contenete le caratteristiche richieste per lo scambio di informazioni;
* rx = lo “UART RX CHAR UUID” che identifica la caratteristica di lettura del server (il robot);
* tx = lo “UART TX CHAR UUID” che identifica la caratteristica di scrittura del server.

Inoltre, un’istanza dell’oggetto Connection presenta due ulteriori attributi inizializzati alla sua creazione:

* client = un’istanza dell’oggetto BleakClient appartenente alla libreria Bleak, identifica un qualsiasi dispositivo connesso tramite BLE;
* queue = un’istanza dell’oggetto QUEUE della libreria Asyncio che rappresenta un buffer in ingresso del client utilizzato per ricevere i dati dal robot.

L’oggetto “Connection” presenta anche diversi metodi asincroni utilizzati per gestire la connessione e il flusso di dati:

* i metodi getter e setter degli attributi con cui l’istanza dell’oggetto viene creata;
* connect = un metodo che ricerca il dispositivo attraverso il suo nome e ne crea un’istanza di BleakClient che ne sancisce la connessione;
* disconnect = il metodo disconnette un dispositivo dal computer;
* send\_message = il metodo scrive sulla caratteristica di lettura del robot, serve per mandare messaggi al dispositivo connesso sottoforma di bytes;
* handle\_rx = un metodo che viene utilizzato ogni qual volta un messaggio viene mandato dal robot al computer, esso ha il compito di inserire nell’attributo queue il messaggio ricevuto;
* getData = un metodo utilizzato per leggere i valori presenti nell’attributo queue e ritornarli sottoforma di stringa;
* hub\_filter = un metodo che imposta l’attributo nome come filtro di ricerca di un dispositivo.

**Pybricks Firmware and Pybricksdev**

Come precedentemente annunciato, per controllare Spike e utilizzarlo all’interno dei nostri algoritmi MARL, abbiamo preferito non utilizzare il firmware base fornito da LEGO, il quale limitava le nostre possibilità, bensì un firmware open-source di terze parti chiamato Pybricks, il quale fornisce codice Python per diversi tipi di Hub Intelligenti LEGO®, tra cui quello di nostro interesse. Esso presenta sia un suo IDE proprietario chiamato Pybricks Code, simile a quello fornito da LEGO® stessa, dove poter scrivere ed eseguire script Python direttamente sull’hub; ed un pacchetto di risorse per sviluppatori, Pybricksdev, contenente sia command-line tools che una libreria per chiamare funzioni equivalenti a quelle chiamabili in Pybricks Code all’interno di un qualsiasi script Python.

Quest’ultimo, principalmente, ci permette di integrare una libreria di funzioni legate al robot con le altre librerie di Python, utili per rendere il codice più snello ed efficiente e ci consente di poter inviare messaggi verso il GATT client e leggere quelli ricevuti dallo stesso.

Pybricksdev si presenta come una libreria contenete diverse classi, ciascuna atta a sfruttare diverse potenzialità dell’ambiente MicroPyhon e del robot. Di seguito andremo a fare una breve discerna delle stesse, le quali singole funzioni possono essere approfondite direttamente sulla loro documentazione. (rif documentazione)

Le classi sono contenute in diversi moduli, i moduli Pybricks sono:

* hubs: questo modulo contiene le classi atte a controllare i diversi hub LEGO®, tra cui il nostro. La classe PrimeHub permette di accedere a tutte le sue funzionalità, dando la possibilità di accedere al giroscopio, allo speaker, al display, ai bottoni, ma anche alla batteria e ai LED;
* pupdevices: questo modulo contiene tutte le classi atte a controllare e programmare i motori e la sensoristica. Ogni classe presenta metodi getter e setter per interagire con questi elementi;
* iodevices: questo modulo presenta classi legate alla gestione di generici strumenti I/O LEGO®, non utili ai fini della nostra sperimentazione;
* parameters: contiene le classi che definiscono virtualmente elementi utili nella gestione del robot quali i pulsanti, le porte dell’hub, i colori e così via;
* tools: contiene Timing tools utili per il monitoraggio e la gestione del workflow
* robotics: questo modulo contiene la classe DriveBase che permette al programmatore di personalizzare e controllare il movimento di guida di Spike. Questa classe è stata utile nel definire i diversi movimenti che il robot può eseguire;
* geometry: piccolo modulo che introduce un’elementale rappresentazione matriciale all’interno di Pybricks, feature assente nei moduli nativi LEGO®;

Gli altri, invece, si interfacciano direttamente con il sistema operativo in MicroPython:

* micropython: fornisce accesso e controllo a diversi moduli interni del sistema operativo quali memoria e costanti;
* uerrno: definisce tutti i codici d’errore del sistema operativo;
* uio: contains stream objects useful for the I\O;
* ujason: utilizzato per convertire oggetti Python nel formato dati JSON;
* umath: questo modulo contiene diverse funzioni matematiche utili a semplificare la programmazione ed i calcoli;
* urandom: modulo che implementa la generazione di numeri pseudo-random;
* uselect: questo modulo viene utilizzato in combinazione con uio e fornisce oggetti utili ad attendere dati su diverse streams;
* ustruct: modulo utile quando si vogliono convertire valori Python in strutture dati C-like;
* usys: è un subset del modulo sys in Python, contiene due streams create con il modulo uio che rappresentano lo standard input e output del sistema operativo. Inoltre, fornisce un controllo di versione dell’OS.

**Spike’s Actions Set**

La libreria Pybricksdev ci ha permesso di controllare al meglio il nostro robot e di programmare un completo set di azioni utile alla nostra sperimentazione.

All’interno del nostro algoritmo, Spike deve essere in grado di muoversi liberamente in avanti o indietro, verso destra o sinistra e girare su sé stesso di un determinato numero di gradi. Inoltre, per poter interagire con l’ambiente circostante, il robot deve poter utilizzare un sensore di colore ed un sensore di distanza che sono stati appositamente montati su di lui.

Per comprendere al meglio come il robot possa muoversi, è utile spiegare come funziona il suo giroscopio e i suoi motori principali, usati per mettere in funzione le ruote motrici.

**Giroscopio**

Il giroscopio di spike funziona su 3 assi principali, di cui può misurare l’angolo di rotazione e la distanza di spostamento relativi ad un determinato punto, impostato come punto zero, solitamente impostato all’avvio attraverso la calibrazione del sensore.

I tre assi vengono chiamati roll, pitch e yaw; i corrispondenti angoli descrivono la rotazione del sensore, e quindi del robot, lungo ciascun asse.

La misura degli angoli è compresa tra -180° e +180° e il nostro angolo d’interesse e l’angolo di yaw che individua la rotazione del robot attorno all’asse verticale, virtualmente uscente dall’hub. Questo angolo risulta fondamentale per poter gestire la rotazione su sé stesso del robot.

Un metodo specifico della libreria Pybricksdev, relativo all’oggetto DriveBase , chiamato state ritorna un array di dimensione 4 contenente la distanza percorsa, la velocità di guida, l’angolo di rotazione e il turn rate.

Nella figura X e la figura Y possiamo vedere i tre assi di riferimento e un focus dettagliato con l’asse di yaw e il modo in cui viene misurato l’angolo.

**Ruote Motrici**

Le ruote motrici vengono azionate dai rispettivi motori angolari a cui sono collegate. Per poterle usare, è necessario creare un’istanza dell’oggetto DriveBase in cui vanno specificati quali sono i due motori che fungeranno da carro motrice, il diametro delle ruote motrici e la distanza tra i punti d’appoggio delle due ruote, chiamata axle track. La precisione degli ultimi due attributi garantisce un movimento coerente di Spike, che in caso contrario potrebbe accumulare piccoli errori sia nel muoversi in linea retta sia nelle curve.

L’oggetto DriveBase ti permette di impostare le impostazioni di velocità e accelerazione dei motori e contiene i metodi per girare su sé stesso e muoversi di una determinata distanza che sono stati necessari per generare i movimenti finali della struttura.

Purtroppo, diversi fattori influenzano il movimento finale del robot, quali l’attrito della superficie su cui si muove e l’attrito della gomma delle ruote. La precisone dello stesso giroscopio e dell’accelerometro non sono elevate, generando movimenti e rotazioni non sempre precise al millimetro.

**Action Description**

Il set di azioni di cui il robot è provvisto è composto dai metodi dell’oggetto Python “Robot”:

* move\_up: il robot attraverso il metodo state
* move\_down
* move\_left
* move\_right
* turn\_right
* turn\_left
* comeback
* back\_in\_the\_hub
* go\_out\_the \_hub
* read\_color

il codice delle diverse azioni è approfondito nell’appendice A.