ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA E SCIENZE INFORMATICHE

Reinforcement Learning per robot grasping in ambiente Gym OpenAI

Tesi in

Machine Learning

Relatore:
Prof. Davide Maltoni

Presentata da:
Giammarco Tosi

Sessione III

Anno Accademico 2015-2016

Indice

Abstract

La teoria del Reinforcement Learning fornisce considerazioni formali, profondamente radicate nelle prospettive psicologiche e neuroscientifiche sul comportamento animale, di come gli agenti siano in grado di ottimizzare il loro controllo su un ambiente. In seguito allo studio dei metodi di Reinforcement Learning, l'obiettivo principale del lavoro di tesi è dimostrare che mediante gli stessi, un agente, servendosi di un braccio meccanico, è in grado di apprendere autonomamente come afferrare un oggetto. Al fine di conseguire l'obiettivo posto vengono realizzati gli ambienti di simulazione 2D robotArm, in diverse configurazioni, e due agenti: Q-Learning e Deep Q-Learning. Il primo agente implementa il metodo Q-Learning come proposto in letteratura con materializzazione della tabella per la stima della funzione valore, mentre il secondo sostituisce la tabella con una rete neurale volta all'approssimazione della funzione valore, la quale ha permesso l'utilizzo di una più dettagliata rappresentazione dello stato dell'ambiente. In seguito ai risultati ottenuti, viene dimostrato che mediante i metodi di Reinforcement Learning, un agente è in grado di apprendere in modo autonomo le dinamiche richieste per afferrare correttamente sia un oggetto in posizione statica che in movimento.

Introduzione

L'obiettivo principale della ricerca in machine learning è fornire ai calcolatori l'abilità di apprendere automaticamente un comportamento sulla base di informazioni ed esempi, senza essere programmati esplicitamente per svolgere un determinato compito. Il Reinforcement Learning, o Apprendimento per Rinforzo, rappresenta un problema di learning riguardante l'apprendimento automatico delle dinamiche di un ambiente, più precisamente, i metodi di Reinforcement Learning cercano di determinare come un agente debba scegliere le azioni da eseguire, dato lo stato corrente dell'ambiente nel quale è situato, con l'obiettivo di massimizzare una sorta di ricompensa totale. L'obiettivo principale del lavoro di tesi è determinare se mediante i metodi di Reinforcement Learning, un agente, servendosi di un braccio meccanico, è in grado di apprendere autonomamente come afferrare un oggetto. In seguito all'obiettivo principale si vogliono confrontare due diverse tipologie del metodo Q-Learning: versione base con materializzazione della tabella per la stima della funzione valore, e versione più avanzata Deep Q-Learning dove la tabella viene sostituita da una rete neurale.

Al fine di conseguire gli obiettivi posti, sono stati realizzati gli ambienti 2D robotArm in due diverse configurazioni: nella prima configurazione, l'oggetto da afferrare assume posizioni statiche casuali nei due assi X e Y ad ogni episodio, mentre nella seconda l'oggetto è in continuo movimento con possibile aumento della difficoltà rendendo casuale la posizione iniziale del braccio meccanico, e la posizione iniziale dell'oggetto. In seguito alle specifiche degli ambienti emerse in fase progettuale, sono state scelte due tecnologie software per la loro realizzazione: OpenAI Gym e Box2D. OpenAI Gym è un toolkit nato per la ricerca in Reinforcement Learning che, oltre ad includere una collezione in continua crescita di ambienti, fornisce interfacce e astrazioni per la definizione di nuovi ambienti. La seconda tecnologie software adottata, Box2D, è una libreria di simulazione 2D di corpi solidi. Per l'implementazione degli ambienti robotArm, OpenAI Gym viene utilizzato come framework, in quanto fornisce le astrazioni, le interfacce per l'implementazione dell'ambiente, e la gestione del rendering, mentre Box2D ha il compito di gestire le dinamiche e la fisica dei componenti dell'ambiente, rendendo la simulazione il più possibile realistica.

Per determinare la bontà dei metodi Q-Learning è stato opportuno implementare i due metodi Q-Learning e Deep Q-Learning. Il primo metodo, Q-Learning con materializzazione della Tabella Q, impone l'utilizzo di uno stato dell'ambiente ridotto e discretizzato, in quanto il metodo prevede l'allocazione in memoria di ogni possibile coppia stato-azione. Il primo metodo, infatti, dato uno stato s, osservando i valori memorizzati in memoria per ogni azione $a_i \in \mathcal{A}(s)$ sceglie di eseguire l'azione contenente il valore maggiore. Discretizzando e riducendo la rappresentazione dello stato, si perdono importanti proprietà sullo stato degli ambienti, in quanto stessi parametri di stato risultano in configurazioni dell'ambiente diverse. Per evitare la perdita di informazioni sulla rappresentazione dello stato dell'ambiente, viene implementato un secondo metodo, Deep Q-Learning, il quale combina Q-Learning, metodo di apprendimento per rinforzo, con un metodo di approssimazione di funzione. Associare una rete neurale a Q-Learning, permette di utilizzare l'intera rappresentazione dello stato senza richiedere riduzione e discretizzazione negli input. Deep Q-Learning sfrutta la rete neurale per approssimare la funzione valore dell'ambiente, ovvero, dato lo stato corrente dell'ambiente in input alla rete neurale, essa ha il compito di stimare i valori delle possibili coppie stato-azione. In questo caso si cerca di far stimare il comportamento della funzione valore alla rete neurale mediante la modifica dei suoi pesi interni, per questo motivo, la configurazione della rete e le sue dimensioni non variano in funzione del numero di valori assumibili dallo stato, permettendo l'uso di valori continui.

In seguito alla realizzazione degli elementi richiesti dal reinforcement learning, vengono misurate le performance degli agenti nell'apprendimento delle dinamiche degli ambienti robotArm. Le performance degli agenti vengono calcolare mediante le migliori performance di accuratezza e ricompensa totale ottenute in media dagli agenti in fase di training, le performance di accuratezza e ricompensa totale ottenute al termine del training, e il tempo complessivo richiesto dagli agenti per completare il training. Inoltre per ogni test eseguito, vengono riportati due grafici: il primo mostra l'andamento delle ricompense totali medie in funzione del numero di episodi portati a termine in fase di training, mentre il secondo mostra la percentuale di episodi completati correttamente in funzione del numero di episodi portati a termine, dove per episodio completato si intende un episodio in cui l'agente ha afferrato l'oggetto con successo entro il tempo limite stabilito.

Nella prima sezione in seguito alla definizione dei fondamenti del machine learning viene introdotto il problema di reinforcement learning. Nella seconda sezione vengono mostrati e messi a confronto i diversi metodi di reinforcement learning a partire dai principi della programmazione dinamica fino ai metodi Temporal Difference con particolare focus su Q-Learning e Deep Q-Learning. Nella terza sezione, in seguito alla progettazione sulla base delle specifiche richieste, viene descritta l'implementazione degli ambienti di simulazione. Nella quarta sezione, viene descritta la realizzazione e le relative scelte implementative degli agenti Q-Learning e Deep Q-Learning mediante richiami al codice. Nella quinta sezione vengono riportati i risultati ottenuti dagli agenti negli ambienti robotArm con relativo confronto dei risultati ottenuti, mentre nella sesta sezione le conclusioni e gli sviluppi futuri individuati.

1 Introduzione al Reinforcement Learning

Nel seguito del capitolo vengono definiti i fondamenti del Machine Learning, introducendo la natura data-driven dei diversi metodi. Ne segue una classificazione per tipologia di problema di learning per poi spostarsi ad una più generale classificazione per tipologia di apprendimento. Gli ultimi paragrafi introducono il Reinforcement Learning, classificandolo come una diversa tipologia di apprendimento del Machine Learning, e descrivendone gli elementi chiave tra cui il processo decisionale di Markov, il quale esprime formalmente l'entità *ambiente* per il reinforcement learning.

1.1 Machine Learning

Con il termine Machine Learning (ML) si identifica un campo della Computer Science che ha lo scopo di fornire ai calcolatori l'abilità di imparare, senza essere programmati esplicitamente per svolgere un determinato compito. Machine Learning esplora lo studio e la costruzione di algoritmi in grado di apprendere dai dati e fare predizioni. Oggi giorno Machine Learning è ritenuto uno degli approcci più importanti dell'intelligenza artificiale, questo perché permette di migliorare ed evolvere il comportamento mediante l'apprendimento di informazioni direttamente dai dati, evitando la programmazione esplicita del comportamento e di conseguenza semplificando lo sviluppo di applicazioni. L'idea alla base di tutti gli algoritmi noti del Machine Learning è il miglioramento automatico del comportamento mediante l'esperienza accumulata in fase di addestramento. Più formalmente "Si dice che un programma **impara** da una esperienza *E* rispetto ad un certo compito *T* e misura di prestazione *P*, se la sua prestazione nel portare a termine un compito *T*, misurata da *P*, migliora grazie all'esperienza *E*" [1].

1.1.1 Dati e Pattern

I dati sono l'ingrediente fondamentale del Machine Learning perché, come già introdotto, il comportamento degli algoritmi non è esplicitamente programmato ma appreso dai dati stessi. Spesso viene utilizzato il termine Pattern per riferirsi ai dati, dove per pattern si intende una regolarità che si riscontra all'interno di un

insieme di dati osservati. La disciplina che studia il riconoscimento dei pattern è il Pattern Recognition la cui intersezione con il Machine Learning è molto ampia ma non completa (il pattern recognition non comprende solo tecniche di Learning ma anche algoritmi esplicitamente programmati).

I pattern possono essere di tipologia numerica o categorica. La prima rappresenta la principale tipologia di dati e si riferisce a valori associati a conteggi o caratteristiche misurabili. I pattern numerici sono tipicamente continui e soggetti a ordinamento. Sono rappresentabili come vettori nello spazio multidimensionale detti anche feature vectors e identificano caratteristiche estratte da segnali come immagini e suoni. I pattern categorici invece si riferiscono a valori associati a caratteristiche qualitative o alla presenza/assenza di una determinata caratteristica (valori booleani). I pattern categorici non sono semanticamente mappabili in valori numerici e sono normalmente gestiti da sistemi basati su regole o alberi di classificazione. I pattern categorici sono maggiormente utilizzati nell'ambito del Data Mining e sono spesso combinati a dati numerici. I pattern possono essere anche di natura sequenziale o ricalcare altre strutture dati complesse come grafi e alberi. Uno stream audio, rappresentante la pronuncia di una parola è ad esempio un pattern sequenziale. I pattern sequenziali sono caratterizzati dalla lunghezza variabile e sono critici da trattare. Richiedono memoria e allineamento spazio-temporale per tener conto del passato, questo perché spesso la posizione nella sequenza e la relazione con i pattern predecessori e successori è di rilevante importanza. Un altro esempio più complesso di pattern è ottenibile mediante la traduzione di una frase in linguaggio naturale dove l'output desiderato è l'insieme degli alberi sintattici plausibili.

1.1.2 Problemi di Learning

Gli algoritmi di machine learning possono essere suddivisi in categorie mediante due diversi principi: per tipologia di apprendimento del "segnale", o per natura dell'output desiderato dal sistema di machine learning. Nel paragrafo corrente vengono descritte le possibili tipologie di problemi di learning suddividendole per natura dell'output desiderato dal sistema.

Classificazione

Con il termine classificazione si definisce l'insieme di algoritmi con lo scopo di assegnare una classe ad un pattern. Gli algoritmi di classificazione o riconoscimento necessitano di apprendere una funzione in grado di eseguire il mapping dallo spazio dei pattern allo spazio delle classi. Si suddividono in due tipologie: classificazione binaria e classificazione multi-classe. Con il termine *classe* si identifica un insieme di pattern aventi proprietà comuni i quali dipendono strettamente dall'applicazione. Ad esempio, nel caso di riconoscimento delle lettere dell'alfabeto inglese scritte a mano, la classe A rappresenta tutti i diversi modi in cui può essere scritto a mano libera il carattere A, mentre il numero totale di classi è pari a 26, una classe per ogni termine dell'alfabeto. Un esempio reale di classificazione è lo Spam Filtering System dove gli input sono i messaggi di posta elettronica e le classi sono "spam" o "non spam". In questo caso si tratta di classificazione binaria mentre nel primo esempio la classificazione è multi-classe.

Regressione

Con il termine regressione si definisce l'insieme di algoritmi con lo scopo di assegnare un valore continuo a un pattern. Questa tipologia di algoritmi di Machine Learning è utile per la predizione di valori continui ed il loro compito si basa sulla risoluzione di un problema di regressione, ovvero, apprendere una funzione che approssima le coppie <input,output> fornite in fase di apprendimento. Ad esempio, la stima dei prezzi di vendita degli appartamenti nel marcato immobiliare rappresenta un problema di regressione.

Clustering

Con il termine Clustering si definisce l'insieme di algoritmi con lo scopo di individuare gruppi di pattern con caratteristiche simili dall'insieme dei dati. In questo caso, il numero di classi del problema non è noto e i pattern non sono etichettati. La natura non supervisionata del problema è ciò che lo rende più complesso rispetto alla classificazione. L'apprendimento dei gruppi di pattern con caratteristiche simili sarà poi utilizzato per determinare le classi del problema. Un esempio di utilizzo del clustering è il raggruppamento di individui sulla base di analogie nel DNA o la segmentazione di immagini in visione artificiale.

Riduzione Dimensionalità

Con il termine Riduzione Dimensionalità si definisce l'insieme di algoritmi volti alla riduzione del numero di dimensioni dei pattern in input. Un algoritmo di riduzione delle dimensionalità consiste nell'apprendere un mapping $\mathbb{R}^d \rightarrow \mathbb{R}^k$ con $d > k$. Data la riduzione di dimensioni, è inevitabile perdere parte dell'informazione. L'obiettivo è quello di conservare le informazioni più salienti e caratterizzanti. Questo insieme di algoritmi è molto utile per rendere trattabili problemi con dimensionalità molto elevata. Più il problema ha dimensioni alte e più le informazioni saranno ridondanti e/o instabili. Gli algoritmi di riduzione della dimensionalità sono inoltre utilizzati per visualizzare in 2D o 3D i pattern che originariamente avevano dimensionalità superiore.

Rapresentation Learning

Con il termine Rapresentation Learning si individua l'insieme di algoritmi volti alla elaborazione automatica dei dati forniti in fase di apprendimento, per la scoperta di una migliore rappresentazione degli stessi. Gli algoritmi di Representation Learning cercano di preservare l'informazione nei loro input trasformandola in modo da renderla più utile. Sono solitamente utilizzati come pre-processing ai raw data[1] prima di eseguirne la classificazione. Gran parte delle tecniche di deep learning (come ad esempio le convolutional neural networks) operano in questo modo, utilizzando come input i raw data ed estraendo automaticamente da essi le features necessarie per risolvere il problema di interesse.

1.1.3 Tipologie di Apprendimento

Come già riportato nel paragrafo precedente, è possibile classificare gli algoritmi di Machine Learning mediante due diversi punti di vista. Nel seguito del paragrafo sono individuate le classi di algoritmi di Machine Learning per tipologia di apprendimento del "segnale".

[1] raw data: dati grezzi ottenuti da una fonte, non ancora processati per l'uso.

Apprendimento Supervisionato

La maggior parte degli algoritmi di ML utilizzati in pratica, è basata sull'apprendimento supervisionato. Si dice apprendimento supervisionato quando si fornisce all'algoritmo un insieme di coppie (X, y), dove X rappresenta un feature vector e y l'output desiderato. L'insieme di coppie (X, y) è utilizzato dall'algoritmo per apprendere una funzione di mapping $y = f(X)$. È chiamato apprendimento supervisionato perché il processo di un algoritmo che apprende da un dataset di training può essere visto come un'insegnante che supervisiona il processo di apprendimento.

Apprendimento Non Supervisionato

Si dice apprendimento non supervisionato quando si fornisce all'algoritmo solo l'insieme di feature vector X sprovvisto degli output desiderati y. L'obiettivo dell'apprendimento non supervisionato è modellare una struttura o distribuzione sui dati in modo da estrarre informazioni aggregate. Si dice apprendimento non supervisionato, perché a differenza del precedente, l'algoritmo non conosce la risposta esatta, e quindi non c'è la supervisione di un insegnante durante la fase di apprendimento. Gli algoritmi devono autonomamente scoprire ed estrarre informazioni di rilievo per la successiva gestione dai dati.

Apprendimento semi-Supervisionato

Si utilizza un approccio semi-supervisionato quando si ha un dataset di grandi dimensioni dove soltanto un sotto insieme dei feature vector X è provvisto di etichetta y. Molti problemi di Machine Learning reali si collocano in questa tipologia di apprendimento, questo perché può essere costoso e dispendioso in termini di tempo etichettare a mano interi dataset. In certe situazioni l'etichettatura di dataset può richiedere l'accesso ad esperti del dominio applicativo. Si utilizza l'apprendimento semi-supervisionato quando la distribuzione dei pattern non etichettati può aiutare a ottimizzare la regola di classificazione.

1.1.4 Valutazione delle prestazioni

Un importante aspetto del Machine Learning è la valutazione delle prestazioni degli algoritmi. Le modalità di valutazione dipendono fortemente dalla tipologia

dell'algoritmo in esame. Una possibilità consiste nell'utilizzo diretto della funzione obiettivo, in genere però si preferisce utilizzare misure più dirette collegate alla semantica del problema. Ad esempio nel caso di problemi di classificazione, l'accuratezza di classificazione, espressa in percentuale, è solitamente la misura di prestazione più significativa. Per errore di classificazione invece si intende il complemento dell'indice di accuratezza.

$$Accuracy = \frac{Pattern\ Correttamente\ classificati}{Pattern\ Classificati} \quad (1)$$

$$Error = 100\% - Accuracy \quad (2)$$

Nel caso più semplice, si assume che il pattern da classificare appartenga a una delle classi note al problema di classificazione, questa tipologia prende il nome di classificazione a *closed set*. In molti casi reali, invece, i pattern da classificare possono appartenere a una delle classi note, o a nessuna di esse, in questo caso la classificazione si dice a *open set*. In classificazioni ad *open set*, non è possibile perciò determinare l'accuratezza o l'errore di classificazione mediante l'utilizzo delle formule riportate perché non tutti i pattern presenti sono classificabili mediante le classi note. Per ovviare al problema esistono due possibili soluzioni. La prima aggiunge una nuova classe fittizia che rappresenta "il resto del mondo" e nel training set si etichettano gli esempi negativi con la nuova classe inserita. Una seconda soluzione, si basa sull'aggiunta di una *soglia di classificazione* a tal fine di consentire al sistema di non assegnare il pattern se in nessuna delle classi la probabilità di appartenenza non supera la soglia imposta.

1.2 Reinforcement Learning

Reinforcement Learning (RL), letteralmente Apprendimento per Rinforzo significa "imparare cosa fare", o meglio come mappare determinate situazioni a delle possibili azioni eseguibili, in modo da massimizzare la ricompensa ottenuta, sotto forma di segnale numerico (reward). Al sistema, non viene detto quale azione è meglio scegliere, come succede nella maggior parte delle tipologie di Machine Learning, ma dovrà scoprire, mediante ripetute prove, quali azioni permetteranno di ottenere la ricompensa maggiore. Nei casi più interessanti e difficili, le azioni

scelte possono avere effetto non soltanto sulla immediata ricompensa ma anche nelle situazioni e ricompense seguenti. Queste due caratteristiche, ricerca mediante trial-and-error e il ritardo nell'ottenimento di ricompense, sono le caratteristiche più importanti che distinguono il Reinforcement Learning dalle altre tipologie di Learning.

Reinforcement Learning non caratterizza soltanto un metodo di apprendimento ma ne definisce un più generale problema di learning. Qualsiasi metodo adatto per risolvere il problema è quindi considerato un metodo di Reinforcement Learning. L'idea di base è catturare gli aspetti più importanti di un problema reale, facendo interagire un agente in grado di apprendere, con l'ambiente rappresentante il problema reale in modo da fargli raggiungere un obiettivo. Per ottenere ciò è opportuno che l'agente sia in grado di interagire con l'ambiente e osservarne lo stato ad ogni istante. L'agente dovrà essere in grado di eseguire azioni le quali avranno effetto sull'ambiente modificandone lo stato. Inoltre l'agente dovrà avere un obiettivo o più obiettivi relativi allo stato dell'ambiente. La formulazione del problema di learning si basa su questi tre aspetti: osservazione, azione e obiettivo.

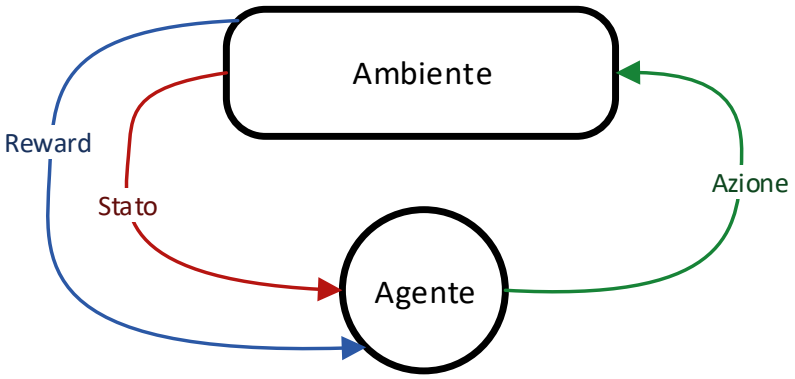

Figura 1 – Interazione tra ambiente e agente nel problema di reinforcement learning

Reinforcement Learning si differenzia dalle altre tipologie di apprendimento 1.1.3. Ad esempio, l'apprendimento supervisionato rappresenta metodi basati sull'apprendimento da esempi pre-classificati, forniti da un supervisore esterno. L'apprendimento supervisionato, come già detto è uno dei più importanti metodi di learning, ma da solo, non è adeguato per l'apprendimento basato su interazione con l'ambiente, questo perché è solitamente impossibile ottenere esempi dei

comportamenti desiderati che sono sia corretti, che rappresentativi di tutte le possibili situazioni in cui l'agente deve scegliere quale azione intraprendere. In territori inesplorati, dove ci si aspetta che l'apprendimento sia la cosa più importante e benefica, un agente deve essere in grado di imparare dalla sua propria esperienza.

Una delle sfide che deriva dal reinforcement learning e non dalle altre tipologia di apprendimento è il bilanciamento tra l'esplorazione di nuove situazioni e lo sfruttamento delle informazioni già apprese (exploration & exploitation). Per ottenere una ricompensa elevata, un agente deve preferire le azioni utilizzate nel passato, che gli hanno consentito di produrre una buona ricompensa. Ma per scoprire tali azioni, l'agente deve scegliere di eseguire azioni che non ha mai provato prima. L'agente deve *sfruttare* quello che già conosce in modo da massimizzare la ricompensa finale, ma allo stesso tempo deve *esplorare* in modo da scegliere azioni migliori nelle esecuzioni future. Il dilemma è che ne l'esplorazione che lo sfruttamento dell'esperienza, scelte esclusivamente, permettono di portare a termine il compito senza fallire. L'agente quindi dovrà provare una varietà di azioni e progressivamente favorire la scelta di quelle che sono apparse come migliori. In un ambiente stocastico, ogni azione dovrà essere provata più volte per ottenere una stima reale della ricompensa prevista.

Un'altra caratteristica chiave del reinforcement learning è che considera esplicitamente l'intero problema dell'interazione tra agente e ambiente, senza concentrarsi su sotto problemi. Tutti gli agenti di RL sono forniti di un obiettivo esplicito, sono in grado di osservare l'ambiente e possono scegliere quale azione intraprendere per influenzare l'ambiente. Inoltre si assume dall'inizio che l'agente dovrà operare ed interagire con l'ambiente nonostante la notevole incertezza nella scelta delle azioni.

1.2.1 Elementi del Reinforcement Learning

Oltre all'agente e l'ambiente, è possibile identificare quattro principali sotto elementi che rappresentano il reinforcement learning: una *policy*, una *reward function*, una *value function* e, eventualmente, un *modello* dell'ambiente.

La *policy* definisce il comportamento che avrà l'agente ad un dato istante in fase di apprendimento. In linea generale, per policy si intende la mappatura tra gli stati osservati dell'ambiente e le azioni che dovranno essere scelte quando l'agente si

troverà in questi stati. Ciò corrisponde a quello che in psicologia è chiamato condizionamento o un set di associazioni stimolo-reazione. In alcuni casi la policy può essere una semplice funzione o una tabella di ricerca (lookup table), mentre in altri casi potrebbe risultare in una estensiva computazione, come ad esempio un processo di ricerca. In ogni caso la policy rappresenta il nucleo dell'agente, nel senso che da sola è sufficiente per determinarne il comportamento.

La *reward function* (o funzione di ricompensa) in un problema di reinforcement learning ne definisce l'obiettivo o goal. Essa mappa ad ogni coppia stato-azione (o meglio ad ogni azione intrapresa in un determinato stato) un singolo numero, detto reward, il quale indica, intrinsecamente, quanto è desiderabile intraprendere una certa azione in un determinato stato. L'obiettivo dell'agente è massimizzare il reward totale ricevuto lungo l'intero periodo. La funzione di reward, definisce la bontà degli eventi per l'agente. In un sistema biologico non sarebbe inappropriato identificare i reward come piacere o dolore. I reward ottenuti nelle coppie stato-azione, rappresentano per l'agente le immediate caratteristiche del problema che sta affrontando. Per questo motivo l'agente non deve essere in grado di alterare la funzione di reward bensì potrà servirsi di essa per alterare la sua policy di comportamento. Ad esempio, se un'azione selezionata dalla policy è seguita da una bassa ricompensa, di conseguenza la policy potrebbe cambiare in modo da scegliere nel futuro in quella stessa situazione, azioni diverse.

Mentre la funzione di reward indica cosa è buono nell'immediato, la *value function* (o funzione di valore) specifica cosa è buono nel lungo periodo. Il valore di uno stato rappresenta la ricompensa totale che l'agente si può aspettare di accumulare nel futuro, partendo da quello stato. Mentre il reward determina l'immediato desiderio di raggiungere uno stato dell'ambiente, il valore ne indica il desiderio a lungo termine considerando non solo lo stato raggiunto nell'immediato, ma anche tutti i possibili stati seguenti e i reward ottenuti raggiungendoli. Per esempio, uno stato potrebbe sempre rendere una bassa ricompensa ma allo stesso tempo permettere di visitare stati, non visitabili altrimenti, che rendono una ricompensa elevata. Analogamente per gli umani, una ricompensa alta rappresenta il piacere mentre una bassa il dolore. I valori invece, rappresentano un più raffinato e lungimirante giudizio di come saranno soddisfatti o insoddisfatti se l'ambiente si trova in un particolare stato.

I reward quindi rappresentano una ricompensa primaria mentre i valori rappresentano la predizione della ricompensa totale, la quale può essere vista

anche come ricompensa secondaria. Senza i reward non esisterebbero i valori, e l'unico scopo di stimare i valori è ottenere ricompense totali più elevate. Tuttavia, le decisioni saranno prese sulla base dei valori stimati, questo perché l'obiettivo dell'agente è massimizzare la ricompensa totale e non le ricompense immediate. Sfortunatamente determinare i valori è più complicato che determinare i reward, questo perché i secondi vengono forniti all'agente direttamente dall'ambiente, mentre i primi devono essere stimati e ristimati mediante le sequenze di osservazioni dell'agente. Proprio per questo motivo il componente più importante della maggior parte degli algoritmi di reinforcement learning è il metodo per la stima efficiente dei valori.

La maggior parte dei metodi di reinforcement learning è quindi strutturata attorno alla stima della funzione valore, anche se non è strettamente necessario per risolvere alcuni problemi di RL. Per esempio, algoritmi meta-euristici come il genetico ed altri metodi di ottimizzazione funzionale sono stati utilizzati per risolvere problemi di reinforcement learning [2] [3]. Questi metodi ricercano direttamente nello spazio delle policies senza mai considerare la stima di funzioni valore. Questa tipologia di algoritmi prende il nome di *evolutionary methods* perché le loro modalità ricalcano l'evoluzione biologica. Se lo spazio delle policies è sufficientemente ridotto, o può essere strutturato in modo da rendere semplice ottenere buone policies, i metodi evolutivi possono essere validi. Inoltre i metodi evolutivi hanno vantaggi in problemi nel quale l'agente non è in grado di osservare accuratamente lo stato dell'ambiente. Tuttavia i metodi basati sull'apprendimento mediante interazione con l'ambiente, in molti casi sono più vantaggiosi dei metodi evolutivi. Questo perché a differenza dei metodi basati su interazione, gli evolutivi ignorano la maggior parte della formulazione del problema di RL: non sfruttano il fatto che la policy che stanno cercando è una funzione che mappa gli stati osservati in azioni da intraprendere. Quando l'agente è in grado di percepire e osservare lo stato dell'ambiente, i metodi ad interazione permettono una ricerca più efficiente.

Il quarto ed ultimo elemento, per alcuni sistemi di reinforcement learning è il *modello* dell'ambiente. Per *modello* si intende un'entità in grado di simulare il comportamento dell'ambiente. Per esempio, dato uno stato e un'azione, il modello è il grado di predire il risultato del prossimo stato e prossimo reward. I modelli sono utilizzati per la pianificazione, dove per pianificazione si intende una qualsiasi modalità di decisione basata su possibili situazioni future, prima che esse siano realmente raggiunte.

1.2.2 Markov Decision Process

Come già definito nei paragrafi precedenti, l'agente in fase di apprendimento basa le proprie decisioni sullo stato percepito dell'ambiente. Nel seguito viene definita formalmente una proprietà degli ambienti e dei loro stati chiamata proprietà di Markov. Per mantenere le formule matematiche semplici, si assume che stati e valori di reward siano finiti. Questo permetterà di definire le formule in termini di somme di probabilità al posto di integrali e densità di probabilità.

Nel caso generale, lo stato dell'ambiente all'istante $t+1$ dopo l'esecuzione dell'azione A_t intrapresa all'istante t, è definibile mediante la distribuzione di probabilità:

$$P\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}, \qquad (3)$$

Per ogni r, s', e ogni possibile valore assumibile dagli eventi passati: $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. Se l'insieme finito di stati assumibili da un ambiente, gode della proprietà di Markov, allora la risposta data dall'ambiente all'istante $t+1$ dipende solo e soltanto dallo stato e dall'azione rappresentanti l'istante t. In questo caso, le dinamiche dell'ambiente possono essere definite mediante la distribuzione di probabilità:

$$P\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\}, \qquad (4)$$

per ogni r, s', S_t, A_t. In altre parole, un'ambiente gode della proprietà di Markov, se e solo se (3) equivale a (4) per ogni r, s', e ogni possibile sequenza di eventi passati $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$.

Se un ambiente gode della proprietà di Markov, allora grazie a (4) è possibile predire il prossimo stato e il prossimo reward conoscendo soltanto la coppia stato-azione dell'istante precedente. Per questo motivo è possibile considerare (4) come la base per la scelta delle azioni da intraprendere. In poche parole, la migliore policy che basa la scelta delle azioni considerando la proprietà di Markov, è buona quanto la migliore policy che basa la scelta delle azioni considerando l'intera sequenza di eventi passati.

Anche quando l'ambiente non gode della proprietà di Markov, è comunque appropriato pensare allo stato nel reinforcement learning come un'approssimazione di uno stato di Markov. La proprietà di Markov è importante nel reinforcement learning perché tutti i metodi legati all'apprendimento per rinforzo, basano le proprie scelte assumendo che i valori forniti dall'ambiente siano in funzione di solo e soltanto lo stato corrente e l'azione intrapresa all'istante precedente. Un'istanza di reinforcement learning che soddisfa la proprietà di Markov è chiamata *Markov Decision Process* (MDP). Se l'insieme degli stati e l'insieme delle azioni sono finite, è detto *finite Markov Decision Process* (finite MDP). Gli MDP finiti, sono molto importanti per la definizione teorica dell'apprendimento per rinforzo. Un finite MDP, è definito mediante un set di stati e azioni e le dinamiche a singolo step dell'ambiente. Data una coppia stato-azione, s e a, la probabilità di ogni possibile prossimo stato, s', è

$$p(s'|s,a) = P\{S_{t+1} = s' | S_t = s, A_t = a\}. \quad (5)$$

La probabilità condizionata (5) viene detta probabilità di transizione. Analogamente, data una coppia stato-azione s e a, combinata ad un qualsiasi stato seguente s', il reward è definibile mediante:

$$r(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']. \quad (6)$$

Questi valori, $p(s'|s,a)$ e $r(s,a,s')$, danno una completa definizione dei più importanti aspetti delle dinamiche di un MDP finito.

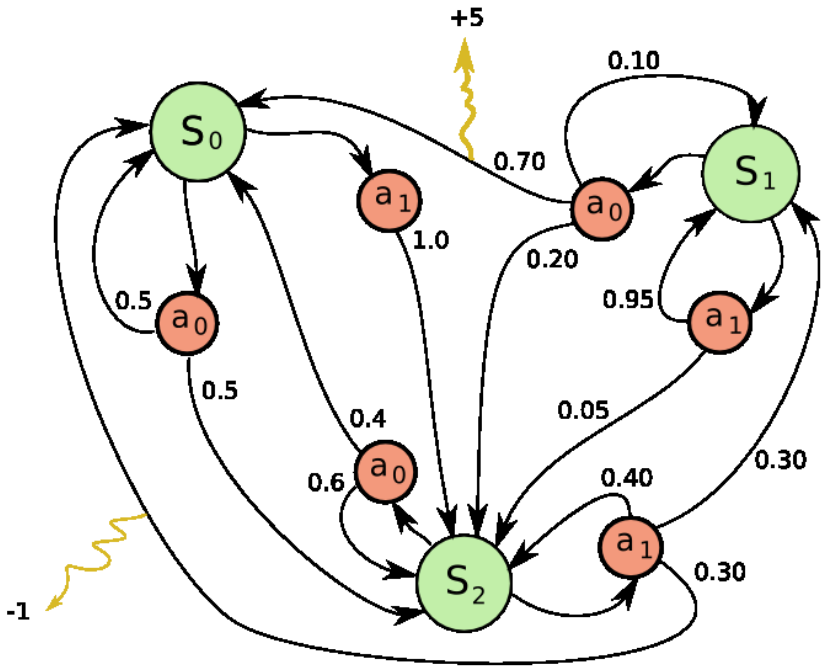

Figura 2 - Esempio di finite MDP con tre stati {S_0, S_1, S_2} e due azioni {a_0, a_1}

La Figura 2 illustra un processo decisionale di Markov composto da un set finito di stati S_i e un set finito di azioni a_j. Ad ogni stato è possibile eseguire una delle possibili azioni, in questo caso la scelta è binaria, a_0 o a_1. In figura è possibile osservare come l'esecuzione di un'azione in un dato stato è seguita da una o al più i transizioni, con $i = 3$ nell'MDP in esame. Ad ogni transizione in figura, è associato un valore rappresentante $p(s_{i+1}|s_i, a_j)$. Trattandosi di una probabilità, la somma di probabilità delle transizioni uscenti da una qualsiasi coppia S_i, a_k deve essere pari a 1. Le transizioni possono avere inoltre un secondo valore associato, il quale ne rappresenta $r(s_i, a_j, s_{i+1})$.

2 Q-Learning

Esistono diversi metodi per risolvere problemi di reinforcement learning. Alla base di ogni metodo è possibile individuarne le idee centrali in comune tra tutti. Mettendo a confronto i metodi si notano diversi approcci all'apprendimento della funzione valore, in ogni caso, l'idea alla base rimane la medesima per tutti i metodi ed è chiamata *Generalized Policy Iteration* (GPI). GPI rappresenta l'approccio iterativo volto alla approssimazione delle funzioni di policy e valore: la funzione valore viene ripetutamente alterata in modo da approssimare la funzione valore relativa alla policy, e la policy è ripetutamente migliorata rispetto alla funzione valore corrente. Nel seguito del capitolo è riportata una dettagliata definizione di GPI.

Con il termine Q-Learning [4], si identifica un metodo off-policy, rappresentante uno dei più importanti passi avanti per la risoluzione di problemi di reinforcement learning. Q-Learning è un metodo Temporal-Difference (TD), una combinazione tra le idee alla base dei metodi Monte Carlo, e le idee alla base della programmazione dinamica (DP). Come i metodi DP, i metodi TD aggiornano le stime utilizzando in parte le stime apprese nelle iterazioni passate. come i metodi Monte Carlo, i metodi TD sono in grado di apprendere direttamente dalle esperienze passate senza il bisogno di un modello delle dinamiche dell'ambiente. Nel seguito del capitolo, dopo aver definito in modo più esaustivo la GPI, vengono presentate le diverse tipologie di approcci al reinforcement learning sottolineandone pregi e difetti. Negli ultimi paragrafi del capitolo viene prima presentato il metodo Q-Learning nella sua versione più semplice, in seguito la versione deep Q-Learning nella quale si sostituisce alla tabella, una rete neurale.

2.1 Metodi Tabular Action-Value

Nel paragrafo sono introdotte le idee centrali di diverse famiglie di metodi di RL basati sulla stima della funzione valore tramite la materializzazione della tabella Action-Value. Il primo sotto paragrafo descrive la logica iterativa in comune a tutti i metodi che prende il nome di Generalized Policy Iteration. In seguito vengono sommariamente descritte le tre famiglie di metodi definendone pregi e difetti. Questo paragrafo ha il compito di introdurre i metodi Q-Learning e Deep Q-Learning trattati nel dettaglio nei paragrafi 2.2 e 2.3

2.1.1 Generalized Policy Iteration

La policy iteration (o iterazione di policy) consiste nella influenza reciproca di due processi: il primo svolge il compito di rendere la funzione valore v consistente con la policy π corrente, ottenendo v_π (processo detto policy evaluation) mentre il secondo ha il compito di modificare la policy seguendo, in modalità greedy[2], i valori estratti dalla funzione valore, $\pi \rightarrow greedy(v)$ (processo detto policy improvement). Nella iterazione di policy generalizzata, questi due processi si alternano, il secondo parte quando il primo termina, anche se ciò non è per forza necessario, esistono varianti nelle quali i processi terminano parzialmente prima di far partire i seguenti. Ad esempio nei metodi Temporal Difference, il processo di policy evaluation ad ogni iterazione aggiorna il valore di una singola coppia stato-azione prima di terminare e consentire l'esecuzione del processo di policy improvement. Se i due processi iterativamente aggiornano tutti gli stati, il risultato finale equivale alla convergenza con la value function ottima v_* e con la politica di scelta delle azioni ottima π_*. Il termine *Generalized Policy Iteration* è utilizzato per riferirsi all'idea generale basata sull'iterazione tra i due processi, indipendentemente dalla granularità di essi. Mediante la GPI è possibile descrivere il comportamento di tutti gli algoritmi trattati nel seguito, e la maggior parte dei metodi di reinforcement learning esistenti. Ciò significa che nella maggior parte dei metodi, è possibile identificare una politica di scelta delle azioni e una funzione valore, dove la prima è sempre migliorata rispetto ai valori stimati dalla seconda, e la seconda è sempre guidata dalla prima, per il calcolo della nuova funzione valore. Quando entrambi i processi si stabilizzano – le iterazioni successive dei processi terminano con risultati equivalenti, o quasi, alle precedenti iterazioni – allora la funzione valore e la policy ottenute dovranno essere ottime. Questo perché la value function si stabilizzerà solo quando sarà consistente con la policy, e la policy si stabilizzerà solo quando avrà un comportamento che segue la value function corrente. Se la policy variasse il comportamento rispetto alla nuova value function, allora di conseguenza anche la value function sarebbe modificata nella iterazione seguente, per modellare meglio il comportamento della policy. Per questo motivo, se la value function e la policy si stabilizzano, saranno sia ottime, che consistenti l'una con l'altra. Il processo di valutazione della funzione valore e il processo di

[2] greedy: termine in informatica utilizzato per rappresentare una famiglia di algoritmi che cercano di ottenere una soluzione ottima globale, attraverso scelte ottime locali.

miglioramento della politica di scelta delle azioni della GPI, sono contemporaneamente in competizione e in cooperazione tra loro. Sono in competizione perché, rendendo la politica greedy rispetto alla funzione valore, tipicamente si rende la value function incorretta per la nuova politica ottenuta, mentre rendendo la value function consistente con la politica, si rende la politica non più greedy rispetto la nuova value function calcolata. Nel lungo andare, in ogni caso, questi due processi interagiscono per trovare una soluzione coincidente tra di essi, la quale equivale all'ottenerne gli ottimi.

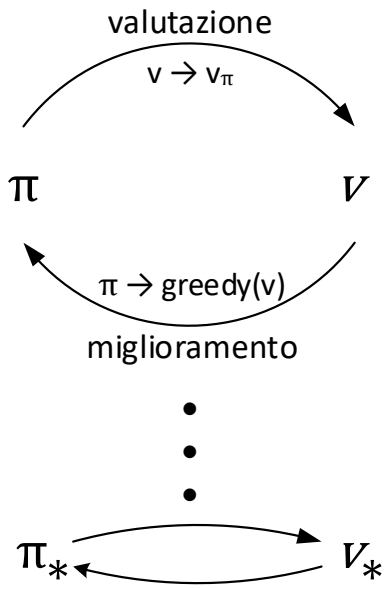

Figure 3 – Generalized policy iteration: La funzione valore e la policy interagiscono fino al raggiungimento della ottimalità e consistenza reciproca.

Per definire la Generalized Policy Iteration sono state citate la funzione valore e la policy, cui ruolo è già stato introdotto nel capitolo precedente, senza però entrare nel dettaglio di come vengono calcolate. Nel seguito sono definite le dinamiche della policy, che come anticipato sono di carattere greedy, e il calcolo e la proprietà fondamentale della funzione valore.

In modo da comprendere pienamente il ruolo della policy e della funzione valore, è bene ricapitolare brevemente gli elementi del problema di RL. L'agente e l'ambiente interagiscono in una sequenza di step discreti nel tempo. Le azioni intraprese nell'ambiente, sono scelte dall'agente. Gli stati sono la base sul quale

l'agente sceglie le azioni da intraprendere, e i reward (o ricompense) sono l'informazione di base per determinare la bontà dell'azione eseguita dall'agente in un determinato stato dell'ambiente. Tutto ciò che si trova all'interno dell'agente, è completamente controllabile da esso, mentre ciò che è all'esterno non è controllabile dall'agente e potrebbe essere in parte non conosciuto (stato dell'ambiente parzialmente assente).

La *policy* è una regola stocastica mediante la quale l'agente seleziona l'azione in funzione dello stato corrente. Dato che l'obiettivo dell'agente è massimizzare la quantità totale di reward ottenuta nel tempo, la *policy* dovrà essere greedy rispetto alla funzione valore, preferendo la scelta di azioni stimate come migliori dalla funzione valore, dove per migliore azione, si intende l'azione con valore maggiore in un dato stato.

La *value function* assegna ad ogni stato, o ad ogni coppia stato-azione, (dipende dalla granularità delle iterazioni, variabile tra i diversi metodi) il rendimento atteso, detto expected return. Per rendimento atteso, si intende la quantità totale di reward ottenibile nel seguito della visita di uno stato. Al raggiungimento della *value function ottima*, ogni stato, o coppia stato-azione, avrà assegnato il maggiore rendimento atteso ottenibile seguendo la policy ottima.

È possibile stimare la *value function* dall'esperienza dell'agente. Ad esempio, se un agente segue una policy π e mantiene in memoria i reward ottenuti per ogni stato incontrato in seguito alla visita di uno stato s, potrà combinarli per ottenere il rendimento atteso (reward totale) dello stato s. Se l'agente mantiene una media del reward totale per s per tutte le volte che lo ha visitato, con il numero di volte tendente ad infinito, la media equivarrà al massimo rendimento atteso ottenibile visitando lo stato s (expected value). Per i MDP, $v_\pi(s)$ è definibile formalmente come:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \,|S_t = s], \tag{7}$$

Dove $\mathbb{E}_\pi[G_t \,|S_t = s]$ rappresenta l'expected value ottenuta dall'agente seguente la policy π, in un qualsiasi step t. Si noti che per definizione, il valore di uno stato terminale è sempre pari a 0. La funzione (7) è detta funzione di stato-valore per la policy π. Similmente, è definibile formalmente una funzione valore più granulare la

quale definisce il valore della scelta di una azione a in uno stato s, mediante una policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \,|S_t = s, A_t = a], \tag{8}$$

q_π prende il nome di funzione azione-valore per la policy π.

È possibile stimare l'*expected return*, ottenibile dopo la visita di uno stato s, mediante la semplice somma dei reward ottenuti nel seguito.

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T, \qquad con\, S_t = s \tag{9}$$

Dove T rappresenta lo step finale, ed S_t lo stato dell'ambiente allo step t. Si assume che nell'ambiente ci sia una naturale nozione di step finale, e quindi che le sequenze di interazione tra ambiente e agente siano suddivise in episodi il quale termine coincide con il raggiungimento di uno dei possibili stati terminali dell'ambiente. Dato che l'ambiente è di natura stocastica, non è possibile essere sicuri che in un seguente episodio, visitando s, si ottenga sempre lo stesso G_t. Occorre aggiungere un concetto di *discounting*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_{k=0}^{T} \gamma^k R_{t+k+1}\,, \qquad con\, S_t = s \tag{10}$$

Con γ compreso tra 0 ed 1, chiamato *discount rate*. Il discount rate permette di considerare con meno peso, scelte intraprese nel futuro rispetto alla scelta intrapresa allo step t. Man mano che ci si allontana dallo stato s, i reward ottenuti hanno un peso sempre inferiore nel calcolo dell'expected return.

Grazie alla definizione di expected return e value function è ora possibile determinare una proprietà fondamentale delle funzioni valore la quale dimostra che esse soddisfano particolari relazioni ricorsive. Per qualsiasi policy π e qualsiasi stato s, è soddisfatta la seguente condizione di consistenza tra il valore di s e il valore di un suo possibile stato successore s':

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{T} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{T} \gamma^k R_{t+k+2} \,\middle|\, S_t = s\right] \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\,[R_{t+1} + \gamma v_\pi(s')], \qquad (11)
\end{aligned}
$$

L'equazione (11) è detta equazione di Bellman per v_π ed esprime la relazione tra il valore di uno stato e il valore degli stati a lui successori. L'equazione di Bellman (11) fa una media fra tutte le possibilità, pesando ognuna di esse con la relativa probabilità. Essa afferma che il valore dello stato s, deve essere equivalente al valore dello stato seguente, ridotto rispetto al parametro γ, sommato al reward ottenuto eseguendo la transizione. In (11), $\pi(a|s)$ rappresenta la probabilità di scegliere l'azione a dato lo stato s. $p(s'|s,a)$ equivale a (5), detta probabilità di transizione allo stato s'. Dato che la condizione di consistenza è soddisfatta da tutte le possibili policy π, occorre generalizzare l'equazione ponendo:

$$
\mathbb{E}_\pi[\cdot] = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\,[\cdot].
$$

Più semplicemente, il valore di uno stato è dato dalla somma degli expected return, pesata per le probabilità di tutte le combinazioni tra scelte di a da parte di π, e possibili stati seguenti s' derivanti dalla natura stocastica dell'ambiente.

L'equazione di Bellman rappresenta la base per il calcolo, l'approssimazione e l'apprendimento della funzione valore.

2.1.2 Programmazione Dinamica

La famiglia di algoritmi chiamata programmazione dinamica (DP), è stata introdotta da Bellman (1957), il quale ha mostrato come questi metodi siano utilizzabili per la risoluzione di una vasta gamma di problemi. Nel seguito viene riportato un sommario di come la programmazione dinamica si approccia alla soluzione di processi decisionali di Markov.

I metodi DP si approcciano alla risoluzione di processi decisionali di Markov mediante l'iterazione di due processi chiamati policy evaluation e policy improvement, come definito nel paragrafo precedente dalla GPI. I metodi DP operano spaziando attraverso l'intero insieme di stati assumibili dall'ambiente,

eseguendo ad ogni iterazione un backup completo per ogni stato. Ogni operazione di aggiornamento eseguita dal backup, aggiorna il valore di uno stato in base ai valori di tutti i possibili stati successori, pesati per la loro probabilità di verificarsi, indotta dalla policy di scelta e dalle dinamiche dell'ambiente. I backup completi sono in stretta relazione con l'equazione di Bellman (11), non sono altro che la trasformazione dell'equazione in istruzioni di assegnazione. Quando una iterazione di backup completo non porta nessuna modifica ai valori degli stati, si ottiene la convergenza e quindi i valori finali degli stati soddisfano appieno l'equazione di Bellman (11). I metodi DP sono applicabili solo se è presente un modello perfetto dell'ambiante, il quale deve equivalere ad un processo decisionale di Markov. Proprio per questo motivo, gli algoritmi DP, sono di poca utilità nel reinforcement learning, sia per la loro assunzione di un perfetto modello dell'ambiente, che per l'alta e costosa computazione, ma è comunque opportuno citarli perché rappresentano la base teorica del reinforcement learning. Infatti, tutti i metodi di RL, cercano di ottenere lo stesso obiettivo dei metodi DP, solo con costo computazionale inferiore e senza l'assunzione di un modello perfetto dell'ambiente. Anche se i metodi DP non sono pratici per problemi vasti, sono comunque più efficienti di metodi basati sulla ricerca diretta nello spazio delle policy, come ad esempio gli algoritmi genetici citati nel paragrafo 1.2 del capitolo *Introduzione al reinforcement learning*. Infatti i metodi DP convergono alla soluzione ottima con un numero di operazioni polinomiali rispetto al numero di stati n e azioni m, contro al numero di operazioni esponenziali m^n richiesto da metodi basati su ricerca diretta.

I metodi DP aggiornano le stime dei valori degli stati, sulla base delle stime dei valori degli stati successori, ovvero aggiornano le stime sulla base di stime passate. Ciò rappresenta una proprietà speciale, che prende il nome di *bootstrapping*. Diversi metodi di RL eseguono *bootstrapping*, anche metodi che non richiedono un perfetto modello dell'ambiente, come richiedono i metodi DP. Nel paragrafo seguente, viene riportato un sommario delle dinamiche e caratteristiche di metodi che non richiedono un modello dell'ambiente, ma non eseguono bootstrap. Queste due caratteristiche sono separate, ma i metodi più interessanti e funzionali, come Q-Learning, sono in grado di combinarle.

2.1.3 Metodi Monte Carlo

I metodi Monte Carlo per stimare la funzione valore e scoprire politiche ottime, non richiedono la presenza di un modello dell'ambiente. Essi sono in grado di apprendere mediante l'utilizzo della sola *esperienza* dell'agente ovvero da campioni di sequenze di stati, azioni e reward ottenute dalle interazioni tra agente e ambiente. L'esperienza può essere acquisita dall'agente in linea con il processo di apprendimento oppure emulata da un dataset precedentemente popolato. La possibilità di acquisire esperienza durante l'apprendimento (*apprendimento on-line*) è interessante perché permette di ottenere comportamenti ottimi anche in assenza di conoscenza a priori delle dinamiche dell'ambiente. Anche l'apprendimento mediante un dataset di esperienze già popolato può essere interessante, questo perché, se combinato con l'apprendimento on-line, rende possibile il miglioramento automatico di policy indotte da esperienze altrui.

I metodi Monte Carlo, per risolvere i problemi di RL, stimano la funzione valore sulla base della somma totale di reward, ottenuta in media negli episodi passati. Ciò assume che l'esperienza sia divisa in episodi, e che tutti gli episodi siano composti da un numero finito di transizioni. Questo perché nei metodi Monte Carlo, solo una volta portato a termine un episodio avviene la stima dei nuovi valori e la modifica della policy. Come la GPI, i metodi Monte Carlo stimano iterativamente policy e funzione valore. In questo caso però, ogni ciclo di iterazione equivale al completamento di un episodio – le nuove stime di policy e funzione valore avvengono episodio per episodio. Solitamente il termine Monte Carlo viene utilizzato per metodi di stima le quali operazioni coinvolgono componenti casuali, in questo caso, il termine Monte Carlo si riferisce a metodi di RL basati su medie di ricompense totali. A differenza dei metodi DP che calcolano i valori per ogni stato, i metodi Monte Carlo calcolano i valori per ogni coppia stato-azione, questo perché in assenza di un modello, i soli valori di stato non sono sufficienti per decidere quale azione è meglio eseguire in un determinato stato. Occorre esplicitamente stimare il valore di ogni azione per permettere alla policy di eseguire le scelte. Per questo motivo, nei metodi Monte Carlo si cerca di ottenere la funzione valore $q_*(s, a)$. Il processo di valutazione per i valori di stato-azione è basato sulla stima di $q_\pi(s, a)$, ovvero l'*expected return* ottenuto a partire dallo stato s, scegliendo l'azione a, seguendo la policy π. Esistono due principali metodi Monte Carlo, i quali si differenziano per modalità di stima degli expected return. Il metodo *every-visit MC* stima il valore di una coppia stato-azione $q(s, a)$ come la media degli expected

return ottenuti in seguito ad *ogni* visita dello stato s e scelta dell'azione a. Il metodo *first-visit MC*, invece, stima $q(s,a)$ come media degli expected return ottenuti in seguito alla sola *prima* visita, in un dato episodio, dello stato s e azione a.

Rispetto ai metodi DP, risulta molto più semplice e deterministica l'idea alla base dei metodi Monte Carlo. Semplificando le dinamiche di calcolo della funzione valore q, si crea però una complicazione. È possibile che diverse coppie stato-azione rilevanti per il raggiungimento dell'obiettivo, non siano mai visitate seguendo la semplice policy che ad ogni stato seleziona l'azione con expected return maggiore. Infatti se si seguisse la pura policy greedy, il comportamento dell'agente diventerebbe deterministico riducendo a zero il fattore esplorazione.

Mantenere una esplorazione sufficiente è un problema nei metodi Monte Carlo. Non è sufficiente scegliere l'azione stimata essere la migliore in quella situazione, perché così facendo non si conoscerebbero gli expected return ottenibili seguendo azioni alternative, i quali potrebbero portare all'apprendimento di una politica di scelta migliore.

Un approccio per risolvere il problema dell'insufficiente esplorazione, si basa sull'utilizzo di una ϵ-greedy policy dove ϵ rappresenta la probabilità di scegliere casualmente l'azione al posto di seguire la scelta definita migliore dalla funzione valore q. Con l'aggiunta del parametro ϵ, le scelte saranno basate sulla probabilità:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \dfrac{\epsilon}{|A(s)|} & se\ a = a^* \\ \dfrac{\epsilon}{|A(s)|} & se\ a \neq a^* \end{cases} \tag{12}$$

dove a^* rappresenta l'azione migliore stimata dalla funzione valore $q(s,a)$. Ad esempio, se le azioni eseguibili nello stato s sono $|A(s)| = 5$, con $\epsilon = 0.4$ la probabilità di scegliere l'azione stimata migliore è pari a $\pi(a^*|s) = 1 - 0.4 + \frac{0.4}{5} = 0.68$ mentre, con probabilità pari a $1 - \pi(a^*|s) = 0.32$, viene eseguita un'azione sub-ottima, garantendo esplorazione.

Un grande vantaggio dei metodi Monte Carlo, rispetto ai metodi DP, è la caratteristica di focalizzare le stime su un più piccolo sotto-insieme di stati assumibili dall'ambiente. Una regione di speciale interesse, può essere accuratamente valutata senza il bisogno di valutare accuratamente l'intero insieme

degli stati. Inoltre i metodi Monte Carlo differiscono dai metodi DP per due motivi. Per prima cosa i metodi MC apprendono direttamente da campioni di esperienza, e quindi è possibile apprendere le dinamiche dell'ambiente in linea con l'acquisizione dell'esperienza da parte dell'agente in assenza di un modello. Infine i metodi MC non eseguono bootstrap, non stimano l'expected return sulla base di altre stime. Nel paragrafo seguente, saranno considerati metodi che apprendono dall'esperienza, come i metodi Monte Carlo, ma eseguono anche bootstrap, come i metodi DP.

2.1.4 Metodi Temporal Difference

Se si volesse identificare un'idea centrale per il reinforcement learning, sarebbe di sicuro l'apprendimento mediante metodi Temporal Difference. L'apprendimento mediante TD è una combinazione delle idee alla base dei metodi Monte Carlo e della Programmazione Dinamica. Come anticipato nell'introduzione al capitolo 2, i metodi TD apprendono direttamente dall'esperienza in assenza di un modello delle dinamiche dell'ambiente, idea alla base dei metodi Monte Carlo. Inoltre come i metodi DP, i metodi TD aggiornano le proprie stime basando il calcolo in parte su stime passate (eseguono bootstrap). Come già verificato per le altre famiglie di metodi, anche i metodi TD seguono l'idea definita dalla GPI per l'iterazione di policy, variandone la granularità.

Rispetto ai metodi MC che attendono il termine di un episodio prima di stimare gli expected return mediante l'esperienza dell'agente, i metodi TD aggiornano gli expected return ad ogni step. Allo step $t+1$, sono già in grado di aggiornare il valore dello stato S_t, osservando il reward R_{t+1} e utilizzando il valore del nuovo stato visitato $V(S_{t+1})$. Nel caso più semplice e generale, la logica di aggiornamento dei valori è definita mediante:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{13}$$

Dove il parametro α compreso tra 0 e 1, rappresenta l'indice di aggiornamento del valore di stato S_t, rispetto alla nuova stima. Dalla formula (13) si nota che il calcolo del valore di uno stato dipende dal reward ottenuto allo step $t+1$ e dal valore del nuovo stato S_{t+1}, ridotto dal parametro di discount γ, introdotto per la formula

generale di calcolo dell'expected return G_t (10). Si noti che con $\alpha = 1$, (13) equivale all'equazione di Bellman (11).

Come per i metodi Monte Carlo, è importante mantenere un buon rapporto tra l'esplorazione e lo sfruttamento delle informazioni apprese (*exploration – exploitation*). I metodi TD si differenziano in due classi: on-policy e off-policy. Per dare una idea più dettagliata del comportamento generale dei metodi TD viene definito *Sarsa*, metodo TD on-policy.

Come per MC, la funzione valore da stimare è $q(s, a)$, perché il valore complessivo di uno stato $v(s)$ non è sufficiente in assenza di un modello dell'ambiente per permettere alla policy di determinare, dato uno stato, quale azione è meglio eseguire. In questo caso però, a differenza di MC, la stima dei valori avviene step per step seguendo l'equazione di Bellman con parametro di aggiornamento (13) considerando però, al posto di uno stato, la coppia stato-azione:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \qquad (14)$$

Essendo di natura on-policy, Sarsa stima q_π sulla base dei comportamenti della politica π, e allo stesso tempo modifica il comportamento greedy della policy rispetto alle stime aggiornate da q_π. La convergenza di Sarsa, e più in generale di tutti i metodi TD, dipende dalla natura della politica. Nei metodi TD, vengono utilizzate politiche ϵ-greedy dove ϵ, ad ogni step viene decrementata. Ad esempio è possibile scegliere $\epsilon = \frac{1}{epoch}$ dove $epoch$ rappresenta l'episodio corrente, in questo modo ϵ viene ridotta con l'avanzare degli episodi, decrementando man mano l'indice di esplorazione. Di seguito viene riportato in pseudocodice, l'algoritmo Sarsa.

$Init\ Q(s,a), \forall s \in S, a \in \mathcal{A}(s), arbitrarily\ and\ Q(terminal\ state,\cdot) = 0$

$Repeat\ (for\ each\ episode)$:

$Init\ S$

$Choose\ A\ from\ S\ using\ \pi\ \epsilon - greedy\ derived\ from\ Q$

$Repeat\ (\ for\ each\ step\ of\ episode)$:

$Take\ action\ A, observe\ R, S'$

$Choose\ A'\ from\ S'\ using\ \pi\ \epsilon - greedy\ derived\ from\ Q$

$Q(S,A) = Q(S,A) + \alpha[R + \gamma Q(S', A') - Q(S,A)]$

$S = S'$

$A = A'$

$Until\ S\ ==\ terminal\ state$

Figura 4 - pseudocodice dell'algoritmo Sarsa, TD on-policy

Grazie alla combinazione delle caratteristiche dei metodi Monte Carlo e DP, Temporal Difference rappresenta vantaggi rispetto ad entrambi. Rispetto ai metodi basati su programmazione dinamica, i metodi TD non richiedono un modello dell'ambiente in grado di stimare la distribuzione di probabilità dei reward e degli stati seguenti, ciò ne aumenta notevolmente le applicazioni. Rispetto ai metodi Monte Carlo, il vantaggio è dato dal fatto che i metodi TD sono realizzati in modo da seguire appieno l'esperienza acquisita incrementalmente con un apprendimento on-line, che stima i nuovi valori delle coppie stato-azione, step per step. Con i metodi Monte Carlo invece, occorre aspettare la fine di un episodio prima di apportare le modifiche ai valori degli stati. In certe circostanze in cui gli episodi hanno una durata elevata, l'apprendimento episodio per episodio di MC allunga notevolmente i tempi. Inoltre l'uso del bootstrap per la stima dei nuovi valori, permette ai metodi TD di ottenere performance migliori rispetto ai metodi MC.

Nel paragrafo seguente viene presentato nel dettaglio Q-Learning, metodo Temporal Difference off-Policy. È stato scelto di dedicare maggiore importanza a Q-

Learning perché è il metodo scelto per l'implementazione degli agenti nella seconda parte della tesi.

2.2 Tabular Action-Value Q-Learning

Q-Learning è un semplice metodo di apprendimento Temporal Difference. Permette ad un agente di apprendere il comportamento ottimale in un processo decisionale di Markov. Q-Learning, come Sarsa, stima la funzione valore $q(s,a)$ in modo incrementale, aggiornando ad ogni step dell'ambiente il valore della coppia stato-azione, seguendo la logica di aggiornamento della formula generale di stima dei valori per i metodi TD (13). Come già anticipato, Q-Learning, a differenza di Sarsa, ha caratteristiche off-Policy, ovvero, mentre il miglioramento della policy avviene in funzione dei valori stimati da $q(s,a)$, la funzione valore aggiorna le stime seguendo una policy secondaria strettamente greedy: dato uno stato, l'azione scelta è sempre quella che massimizza il valore $\max_a q(s,a)$. La policy π ha comunque un importante ruolo nella stima dei valori perché mediante essa vengono determinate le coppie stato-azione da visitare e aggiornare.

$$Q(S_t,A_t) = Q(S_t,A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)] \qquad (15)$$

La formula (15) rappresenta le modalità di stima delle coppie stato-azione nel metodo Q-Learning. Come in (13) presenta due parametri: γ è il parametro di discounting compreso tra 0 e 1 il quale permette di dare meno peso a reward ottenuti nel futuro, mentre α rappresenta il tasso di apprendimento e come γ, può assumere valori compresi tra 0 e 1. Ponendo $\alpha = 1$, l'aggiornamento del valore di una coppia stato-azione, sovrascrive la stima passata, sostituendola con la nuova stima, mentre con $\alpha = 0.5$ il valore aggiornato della coppia stato-azione risulta pari alla media fra la nuova stima e la stima precedente.

Q-Learning utilizza una policy di tipo ϵ-greedy. Come per Sarsa, ϵ può essere scelta in modo da decrementare l'esplorazione linearmente rispetto agli episodi portati a termine, ad esempio ponendo $\epsilon = \frac{1}{epoch}$.

Grazie alla natura off-policy di Q-Learning, l'analisi dell'algoritmo risulta più semplice perché assume un comportamento indipendente dalla policy di scelta. Infatti, Christopher J.C.H. Watkings, l'ideatore di Q-Learning (1989), in

collaborazione con Peter Dayan nel 1992 hanno dimostrato la convergenza matematica di Q-Learning [5], a differenza di Sarsa, per cui in letteratura non è tutt'ora presente la dimostrazione di convergenza.

Nel caso più semplice Q-Learning si serve di una tabella per memorizzare ogni coppia stato-azione. Ad ogni step, l'agente osserva lo stato corrente dell'ambiente e servendosi della policy π, seleziona ed esegue l'azione. Eseguendo l'azione l'agente ottiene il reward R_{t+1} e il nuovo stato S_{t+1}. A questo punto l'agente è in grado di calcolare $Q(S_t, A_t)$ aggiornandone la stima. Di seguito viene riportato l'algoritmo in pseudocodice.

$Init\ Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s), arbitrarily\ and\ Q(terminal\ state, \cdot) = 0$

$Repeat\ (for\ each\ episode)$:

$Init\ S$

$Repeat\ (for\ each\ step\ of\ episode)$:

$Choose\ A\ from\ S\ using\ \pi\ \epsilon - greedy\ derived\ from\ Q$

$Take\ action\ A, observe\ R, S'$

$$Q(S,A) = Q(S,A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S,A)\right]$$

$S = S'$

$Until\ S == terminal\ state$

Figura 5 - Pseudocodice dell'algoritmo Q-Learning, metodo off-policy Temporal Difference

2.3 Deep Q-Learning

Nei paragrafi precedenti, le stime della funzione valore sono state rappresentate mediante l'utilizzo di una tabella, nella quale ogni casella rappresenta uno stato, o una coppia stato-azione. L'utilizzo di una tabella per rappresentare la funzione valore, permette di realizzare algoritmi semplici e, se gli stati dell'ambiente sono Markoviani, permette di stimare accuratamente la funzione valore perché assegna ad ogni possibile configurazione dall'ambiente, l'expected return appreso durante le iterazioni di policy. L'utilizzo della tabella però porta anche limitazioni, infatti i

metodi tabular action-value sono applicabili solo ad ambienti con ridotto numero di stati e azioni. Il problema, non si limita alla sola quantità elevata di memoria richiesta per memorizzare la tabella, bensì anche all'elevato numero di dati e tempo richiesti per stimare ogni coppia stato-azione in modo accurato. In altre parole, il problema principale è la *generalizzazione*. Occorre trovare una soluzione al problema, generalizzando l'esperienza acquisita su un sotto insieme delle coppie stato-azione, in modo da approssimarne un insieme più ampio. Fortunatamente, la generalizzazione sulla base di esempi è già stata studiata estensivamente e non occorre inventare totalmente nuovi metodi da utilizzare nel reinforcement learning. Le soluzioni alla generalizzazione si basano sulla combinazione dei metodi RL con metodi di *approssimazione di funzione*. Solla base di un sotto insieme di esempi di comportamento di una determinata funzione, i metodi di approssimazione di funzione, cercano di generalizzare rispetto ad essi per ottenere un'approssimazione dell'intera funzione.

Come per i metodi basati su tabella, esistono vari metodi RL di approssimazione di funzione. Per non rendere la trattazione troppo complessa, nel seguito viene introdotto solamente Deep Q-Learning, evoluzione di Q-Learning descritto nel paragrafo precedente.

Con il termine Deep Q-Learning [6] si identifica un metodo RL di approssimazione di funzione. Esso rappresenta quindi un'evoluzione del metodo base Q-Learning in quanto la tabella stato-azione è sostituita da una rete neurale, con lo scopo di approssimare la funzione valore ottima q_*.

Rispetto agli approcci precedenti, dove si utilizzava strutturare la rete in modo da richiedere in input sia stato che azione e fornendo in output il relativo expected return, Deep Q-Learning ne rivoluziona la struttura in modo da richiedere in input solo lo stato dell'ambiente e fornendo in output tanti valori stato-azione quante sono le azioni eseguibili nell'ambiente.

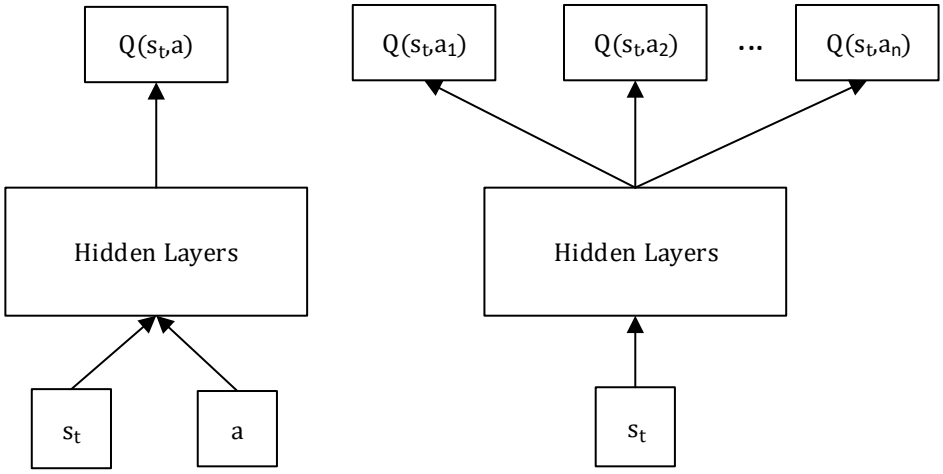

Figure 6 - a sinistra: struttura naive. a destra: struttura ottimizzata utilizzata nel metodo deep Q-Learning

Dato che ad ogni aggiornamento di valore (15) occorre determinare $\max_a q(s,a)$, con la configurazione naive mostrata in Figure 6, ad ogni step è necessario eseguire n passi forward, con $n = |\mathcal{A}(s)|$. Al contrario in deep Q-Learning, il numero di passi forward è sempre pari a 1, qualsiasi sia il numero di azioni eseguibili, perché l'output della rete, è composto da tanti neuroni quante sono le azioni, e il valore contenuto in essi rappresenta l'expected return delle relative azioni.

Nel metodo semplice con tabella, l'apprendimento avviene accedendo alla casella rappresentante la coppia stato-azione e aggiornando l'expected return in base alla nuova stima seguendo la formula (15). Questa modalità di apprendimento, non è applicabile ad una rete neurale in quanto l'unico modo per modificarne il comportamento è attraverso l'aggiustamento dei pesi, eseguendo un passo backward. L'apprendimento della funzione valore nel metodo Deep Q-Learning è basato sull'aggiustamento dei pesi della rete in funzione della *loss function*:

$$L_t = \left[\left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a)\right) - Q(S_t, a_t)\right]^2 \qquad (16)$$

Dove $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ rappresenta il target, ovvero l'expected return ottimo, mentre $Q(S_t, a_t)$ rappresenta il valore stimato dalla rete. Chiaramente, non si è a conoscenza dell'expected return ottimo, per questo motivo occorre stimarlo.

La stima dell'expected return ottimo $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ può avvenire mediante tecniche simili al calcolo dell'expected return utilizzato nei metodi Monte Carlo 2.1.3, oppure utilizzando direttamente la rete. Nel secondo caso, si noti che i valori target dipendono dalla configurazione dei pesi della rete, ai quali saranno apportate modifiche ad ogni step. Dato l'impiego di una loss function, Deep Q-Learning tratta la stima della funzione valore come un problema di regressione. Gli errori calcolati dalla loss function saranno propagati all'indietro nella rete mediante un passo backward, seguendo la logica di discesa del gradiente con l'intento di minimizzare l'errore.

Con le modifiche apportate è ora possibile definire un primo algoritmo Deep Q-Learning base, dove la tabella delle coppie stato-azione è sostituita da una rete neurale inizializzata con pesi casuali e l'apprendimento della funzione valore è ottenuto mediante la minimizzazione degli errori calcolati dalla loss function.

$Init\ Q(s,a), with\ random\ weights\ w$

$Repeat\ (for\ each\ episode)$:

    $Init\ and\ observe\ S$

    $Repeat\ (for\ each\ step\ of\ episode)$:

        $Choose\ A\ from\ S\ using\ \pi\ \epsilon - greedy\ derived\ from\ Q$

        $Take\ action\ A, observe\ R, S'$

        $calculate\ target\ T$

            $if \quad S' \ is\ terminal\ state\ then\ T = R$

            $else \quad T = R + \gamma \max_a Q(S', a)$

        $Train\ the\ Q\ Network\ using\ \left(T - Q(S,A)\right)^2\ as\ loss$

        $S = S'$

    $Until\ S == terminal\ state$

Figura 7 - Deep Q-Learning base con stima del target mediante rete neurale

Utilizzando l'algoritmo base mostrato in Figura 7, si scopre che l'approssimazione della funzione valore mediante rete neurale è tutt'altro che

stabile. Per ottenere la convergenza è opportuno modificare l'algoritmo base introducendo tecniche per evitare oscillazioni e divergenze.

La tecnica più importante prende il nome di *experience replay*. Nel corso degli episodi, ad ogni step, l'esperienza dell'agente $e_t = (s_t, a_t, r_t, s_{t+1})$ viene memorizzata in un dataset $D_t = \{e_1, \dots, e_t\}$, chiamato replay memory. Nel ciclo interno dell'algoritmo, al posto di eseguire il training alla rete in base alla sola transizione appena eseguita, viene selezionato un sotto insieme di transizioni in modo casuale dalla replay memory, ed il training avviene in funzione del loss (es. dell'errore quadratico) calcolato sul sotto insieme di transizioni. Questa tipologia di aggiornamento prende il nome di aggiornamento a minibatch e porta un notevole vantaggio rispetto al metodo base. Per prima cosa, ogni step di esperienza è potenzialmente utilizzato in diversi aggiornamenti dei pesi della rete, e ciò permette di utilizzare i dati in maniera più efficiente. Inoltre, apprendere direttamente da transizioni consecutive è inefficiente per via della forte correlazione tra esse. La tecnica experience replay, mediante la selezione casuale delle transizioni dalla replay memory, elimina il problema della correlazione tra le transizioni consecutive e riduce la varianza tra i diversi aggiornamenti. In fine, l'utilizzo dell'experience replay permette di evitare di convergere in un minimo locale o di divergere catastroficamente, in quanto l'aggiornamento dei pesi si basa sulla media di diversi stati precedenti, smussando l'apprendimento ed evitando oscillazioni o divergenze nei parametri. In pratica l'algoritmo modificato memorizza le ultime n esperienze nella replay memory D, e sorteggia casualmente un sotto insieme di esperienze da D ogni volta che esegue un aggiornamento dei pesi della rete.

$Init\ replay\ memory\ D\ to\ capacity\ N$

$Init\ Q(s,a), with\ random\ weights\ w$

$Repeat\ (for\ each\ episode)$:

$Init\ and\ observe\ S$

$Repeat\ (\ for\ each\ step\ of\ episode)$:

$Choose\ A\ from\ S\ using\ \pi\ \epsilon-greedy\ derived\ from\ Q$

$Take\ action\ A, observe\ R, S'$

$Store\ experience\ (S, A, R, S^{'})\ in\ D$

$Sample\ random\ transition\ (SS, AA, RR, SS^{'}) from\ D$

$calculate\ target\ TT\ foreach\ selected\ transition\ from\ D$

$if \quad S^{'}\ is\ terminal\ state\ then\ TT = RR$

$else \quad TT = RR + \gamma \max_{a} Q(SS^{'}, a)$

$Train\ the\ Q\ Network\ using\ \left(TT - Q(SS, AA)\right)^2\ as\ loss$

$S = S^{'}$

$Until\ S\ ==\ terminal\ state$

Figura 8 - algoritmo in pseudocodice per Deep Q-Learning con experience replay

3 robotArm in ambiente OpenAI Gym

Con il termine RobotArm si identifica una serie di ambienti caratterizzati dalla presenza di un braccio robotico a tre snodi munito di pinza, pilotabile da un agente esterno, e un oggetto da afferrare. Per consentire le dinamiche di interazione tra ambiente e agente previste dai metodi di reinforcement learning, l'ambiente è di natura episodica a step.

Nel seguito del capitolo vengono affrontate le fasi di progettazione e implementazione volte alla creazione dell'ambiente robotArm, il cui scopo è il test e confronto di due implementazioni di metodi per il reinforcement learning. Nel primo paragrafo viene affrontata la progettazione nella quale, oltre a definire globalmente l'ambiente e le sue dinamiche, sono definiti gli elementi richiesti dal reinforcement learning (vedi Sezione 1.2.1), come lo stato, la funzione reward e le dinamiche di interazione con l'agente.

In funzione delle caratteristiche dell'ambiente emerse in fase progettuale, sono state scelte due tecnologie software per lo sviluppo dell'ambiente: OpenAI Gym e Box2D. OpenAI Gym è un toolkit nato per la ricerca in Reinforcement Learning che, oltre ad includere una collezione in continua crescita di ambienti, fornisce interfacce e astrazioni per la definizione di nuovi ambienti. Box2D è una libreria di simulazione 2D di corpi solidi ed è nato per la gestione del motore fisico nei videogames. In breve, per l'implementazione degli ambienti robotArm, OpenAI Gym viene utilizzato come framework, in quanto fornisce le astrazioni, le interfacce per l'implementazione dell'ambiente e la gestione del rendering, mentre Box2D ha il compito di gestire le dinamiche e la fisica dei componenti dell'ambiente, rendendo la simulazione il più possibile realistica. Nell'ultimo paragrafo sono descritte le due tecnologie software adottate e l'implementazione degli ambienti robotArm.

3.1 Progettazione ambienti robotArm

La realizzazione degli ambienti robotArm è volta al test e confronto di diversi metodi di Reinforcement Learning: Q-Learning con tabular Action-Value 2.2 e Deep Q-Learning 2.3. In primo luogo si vuole verificare se mediante l'apprendimento per rinforzo un agente è in grado di apprendere autonomamente come afferrare un oggetto. In secondo luogo si vogliono mettere a confronto i due metodi sopra citati.

La progettazione degli ambienti RobotArm prevede la definizione di diversi elementi del problema di reinforcement learning. Nel seguito del paragrafo sono discusse le specifiche progettuali facendo focus sui diversi elementi relativi all'entità ambiente, definiti nel primo capitolo al paragrafo 1.2.1. Per ognuno degli elementi vengono descritte e discusse le scelte.

Si vogliono realizzare due versioni di robotArm: nella prima versione, l'oggetto da afferrare assume posizioni statiche casuali nei due assi ad ogni episodio, mentre nella seconda l'oggetto è in continuo movimento con possibile aumento della difficoltà rendendo casuale la posizione iniziale del braccio meccanico, e la posizione iniziale dell'oggetto.

Per realizzare *robotArm Simple*, prima versione dell'ambiente, la soluzione adottata prevede l'aggiunta di un piedistallo con altezza e posizione nell'ambiente casuale ad ogni episodio. Per quanto riguarda invece la versione con oggetto in movimento, la soluzione adottata prevede l'utilizzo di un half pipe sul quale l'oggetto è in continuo movimento oscillatorio, la variante prende il nome di *robotArm Pipe*. entrambi gli ambienti sono osservabili in Figura 9.

Per prima cosa, gli ambienti devono consentire l'interazione prevista dal reinforcement learning, osservabile in Figura 1: l'ambiente deve essere progettato in modo da consentire ad un agente esterno l'esecuzione di una azione e di conseguenza restituire all'agente il relativo reward e il nuovo stato.

Il *braccio meccanico* è composto da un basamento statico posizionato sul terreno, che fa da supporto ai tre segmenti che lo compongono. I tre segmenti sono collegati tra loro ed ognuno di essi assume una dimensione circa il 60% rispetto al segmento che lo precede, in modo da ridurre le dimensioni proporzionalmente più il segmento è vicino alla pinza. Ogni giunto tra due segmenti del braccio permette la rotazione di un certo angolo e possiede un motore mediante il quale è possibile guidare il braccio robotico per ottenere la posizione desiderata. La pinza è composta da due branche unite a cerniera in grado di acquisire posizione di

apertura o di chiusura. Ogni branca della pinza è in grado di percepire il contatto con l'oggetto da afferrare, requisito primario per determinare il raggiungimento dell'obiettivo. All'inizio di ogni episodio, la pinza è in posizione aperta.

Dato che gli ambienti devono essere di natura episodica, è opportuno che raggiungano prima o poi uno stato terminale. Ogni episodio deve terminare, sia in caso di raggiungimento dell'obiettivo che in caso contrario. Guidati dalla natura dell'obiettivo dell'ambiente, gli stati terminali possono essere di due tipologie: successo e insuccesso: successo nel caso in cui l'agente riesce ad afferrare l'oggetto, di insuccesso se l'agente urta l'oggetto facendolo cadere al suolo (nel primo caso facendolo cadere dal piedistallo, nel secondo facendolo uscire dall'half pipe). In questo modo però, se l'agente non urta o afferra l'oggetto, l'episodio non termina, per questo motivo è opportuno aggiungere un terzo stato terminale il quale mette conclude l'episodio dopo 500 step di interazione.

Una buona *funzione reward*, è un'importante fattore per il raggiungimento dell'obiettivo, questo perché il goal per un agente rimane l'ottenere il maggior reward totale al termine dell'episodio. Per questo motivo la funzione reward deve essere definita privilegiando l'ottenimento dell'obiettivo finale, evitando di far convergere il comportamento dell'agente in minimi locali. Fornendo un singolo reward positivo al solo completamento, si renderebbe più complicato e lento il raggiungimento dell'obiettivo, per questo motivo occorre inserire reward intermedi più piccoli per indirizzare l'agente verso la scelta di azioni che lo guidino ad ottenere l'obiettivo finale. Oltre a favorire azioni intermedie corrette, occorre allo stesso modo disincentivare azioni che divergono dall'obiettivo restituendo reward negativi all'agente. Un ultimo fattore è dato dalla posizione della pinza. La chiusura preventiva della pinza, porta all'obbligo di riapertura e richiusura una volta raggiungo l'obiettivo. Questo comportamento è scorretto, ed è opportuno tenerne conto in fase di reward. Per questo motivo tutti i reward intermedi in presenza di pinza chiusa, vengono ridotti di una costante. Per ottenere la funzione reward desiderata, ad ogni step l'ambiente calcola la distanza tra la pinza e l'oggetto da afferrare, e mantiene in memoria la distanza calcolata allo step precedente. In questo modo conoscendo le due distanze è possibile definire se grazie all'ultima azione eseguita, l'agente ha avvicinato o allontanato la pinza del braccio meccanico all'oggetto. Se la distanza allo step corrente d_t è minore rispetto alla distanza allo step precedente d_{t-1}, allora l'agente riceve un reward $r_t = +1$. Al contrario, l'agente ottiene un reward $r_t = -1$. Per quanto riguarda la posizione della pinza, il

reward ottenuto viene decrementato o incrementato di un fattore pari a 0.2, per scoraggiare aperture non necessarie della pinza stessa. Più formalmente:

$$r_t = \begin{cases} +1 + 0.2 * statoPinza & se\ d_t < d_{t-1} \\ -1 + 0.2 * statoPinza & altrimenti \end{cases} \quad (17)$$

Con $statoPinza = 0$ quando la pinza è chiusa mentre $statoPinza = 1$ quando la pinza è aperta. Si noti che se l'agente allo step corrente non esegue alcuna azione, mantenendo fermo il braccio, dato che la nuova distanza non risulta minore della precedente, otterrà un reward negativo.

Nell'ambiente RobotArm Simple, versione con oggetto in posizione statica, al completamento dell'obiettivo, l'agente ottiene un reward pari a 200. In questo modo per massimizzare il total reward, l'agente deve cercare di ottenere l'obiettivo evitando azioni scorrette, e quindi raggiungendo l'oggetto nel numero minore di mosse possibili. Infatti la funzione reward, per come è stata realizzata, tiene conto anche del tempo impiegato dall'agente per raggiungere l'obiettivo.

Per quanto riguarda invece RobotArm Pipe, versione con oggetto in movimento oscillante, la funzione reward definita precedentemente non è sufficiente per incentivare l'agente a raggiungere l'obiettivo nel minor numero di step possibili. Questo perché essendo l'oggetto in movimento, in caso di oscillazione in direzione braccio meccanico, pur mantenendo il braccio immobile, la distanza tra pinza e oggetto si ridurrebbe, portando l'agente ad ottenere reward positivi. Per questo motivo per RobotArm Pipe, il reward ottenuto al completamento dell'obiettivo è pari a $r = \frac{500}{oscillazioni}$ dove il denominatore rappresenta il numero di oscillazioni complete eseguite dall'oggetto prima di essere afferrato. Grazie alla modifica applicata alla funzione reward, anche nell'ambiente RobotArm Pipe viene incentivato l'agente a completare più velocemente l'obiettivo.

Definita la funzione reward, è ora opportuno definire lo *stato* dell'ambiente. Con il termine *stato* si intende un vettore di parametri che definisce la configurazione corrente dell'ambiente.

Indice	*Parametro*	*Range*	*Descrizione*
0	Stato pinza	0 – 1	Stato della pinza, aperto o chiuso
1	Angolo spalla	30° – 90°	Valore dell'angolo del giunto tra base e primo segmento del braccio meccanico
2	Angolo gomito	-90° – 0°	Valore dell'angolo del giunto tra primo segmento e secondo segmento del braccio meccanico
3	Angolo polso	-90° – 0°	Valore dell'angolo del giunto tra secondo segmento e terzo segmento del braccio meccanico
4	Delta X	-8 – 20	Distanza sull'asse X tra pinza e oggetto da afferrare
5	Delta Y	-13 – 13	Distanza sull'asse Y tra pinza e oggetto da afferrare

Tabella 1 - Definizione dei parametri scelti per la definizione di stato degli ambienti robotArm

Per gli ambienti realizzati si è scelto di definire lo stato mediante informazioni quali: stato della pinza, angoli dei giunti del braccio, e le distanze nei due assi tra la pinza del braccio e l'oggetto. L'insieme dei parametri scelti è osservabile in Tabella 1. Si noti che tra i parametri che definiscono lo stato, non è presente la posizione dell'oggetto, questo perché si è voluto evitare di inserire posizioni assolute, preferendo distanze relative come la distanza tra la pinza e l'oggetto, informazione che codifica al suo interno sia la posizione della pinza che la posizione dell'oggetto.

Indice	*Azione*	*Descrizione*
0	Azione nulla	Non viene eseguita alcuna azione
1	Incremento spalla	Incremento dell'angolo del giunto tra base e primo segmento del braccio meccanico
2	Decremento spalla	Decremento dell'angolo del giunto tra base e primo segmento del braccio meccanico
3	Incremento gomito	Incremento dell'angolo del giunto tra primo segmento e secondo segmento del braccio meccanico
4	Decremento gomito	Decremento dell'angolo del giunto tra primo segmento e secondo segmento del braccio meccanico
5	Incremento polso	Incremento dell'angolo del giunto tra secondo segmento e terzo segmento del braccio meccanico
6	Decremento polso	Incremento dell'angolo del giunto tra secondo segmento e terzo segmento del braccio meccanico
7	variazione stato pinza	Cambio di stato della pinza del braccio meccanico.

Tabella 2 - Definizione delle possibili azioni eseguibili negli ambienti robotArm

L'ambiente permette otto possibili azioni. Come definito in Tabella 2, per ognuno dei tre giunti del braccio, vengono definite due azioni per incrementare o decrementare l'angolo. La settima azione permette all'agente di modificare lo stato della pinza mentre l'azione con indice 0 rappresenta l'azione nulla.

3.2 Implementazione degli ambienti

In seguito alla completa progettazione degli ambienti sono state individuate due tecnologie software utilizzate come base per l'implementazione: OpenAI Gym e Box2D introdotte nel seguito del paragrafo. In fine viene definita la realizzazione degli ambienti mostrando come le due tecnologie sono state combinate.

3.2.1 OpenAI Gym

OpenAI Gym [7] è un toolkit per la ricerca in Reinforcement Learning scritto in codice Python [8]. Il toolkit include una collezione in continua crescita di ambienti i quali espongono problemi noti in Reinforcement Learning con un'unica comune interfaccia e un website dove ogni utente ha la possibilità di pubblicare i risultati ottenuti su un certo ambiente, in modo da confrontare le performance dei diversi algoritmi con la community. Gli utenti, inoltre, sono incoraggiati a condividere anche il codice sorgente che ha permesso loro di ottenere i risultati caricati, munito di istruzioni nel dettaglio di come riprodurre i risultati da loro ottenuti. OpenAI Gym fornisce inoltre le astrazioni e le interfacce per la realizzazione di nuovi ambienti ed è in grado di gestire il rendering, evitando allo sviluppatore di preoccuparsene. Dato che il framework è nato appositamente per lo studio di algoritmi per il reinforcement learning, le interfacce proposte per le interazioni tra ambiente e agente ricalcano esattamente gli elementi richiesti dal problema. OpenAI Gym presuppone che l'ambiente sia di natura episodica e che l'agente interagisca con esso ad ogni step eseguendo una delle possibili azioni. Interagendo con l'ambiente, l'agente ottiene informazioni quali stato, reward e un flag indicante l'eventuale completamento dell'episodio, utilizzato dall'agente per determinare quando è opportuno resettare l'ambiente ed iniziare un nuovo episodio.

OpenAI Gym mette a disposizione la classe Env, la quale incapsula l'ambiente e le eventuali sue dinamiche interne. La classe presenta diversi metodi e attributi da implementare per realizzare un nuovo ambiente. I metodi più importanti prendono il nome di reset, step e render.

Il metodo *reset* ha il compito di resettare l'ambiente inizializzandolo allo stato iniziale. All'interno del metodo reset devono essere contenute le definizioni degli elementi che compongono l'ambiente, in questo caso, la definizione del braccio meccanico, dell'oggetto da afferrare e relativo supporto.

Il metodo *step* ha il compito di far avanzare di un passo temporale l'ambiente. Esso richiede in ingresso l'azione da eseguire e restituisce la nuova osservazione all'agente. All'interno del metodo devono essere definite la gestione delle dinamiche dei movimenti, il calcolo dello stato e del reward, e i controlli di completamento dell'episodio.

Il terzo ed ultimo metodo è *render* al quale interno deve essere definito come gli elementi ad ogni step devono essere rappresentati. Il metodo prevede diverse

tipologie di rendering come: human, rgb_array o ansi. Con tipologia human, il rendering avviene sullo schermo o terminale e il metodo non restituisce nulla, con tipologia rgb_array, invocando il metodo viene restituito un array n-dimensionale che rappresenta i pixel rgb della schermata, scegliendo la terza tipologia il metodo render restituisce una stringa contenente una rappresentazione testuale. Per eseguire il rendering, OpenAI Gym mette a disposizione la classe viewer mediante la quale è possibile disegnare gli elementi dell'ambiente sotto forma di insieme di poligoni e cerchi.

Per quanto riguarda gli attributi dell'ambiente, Env prevede la definizione di action space, observation space e reward range. L'attributo *action space* rappresenta lo spazio delle azioni, ovvero l'insieme delle possibili azioni eseguibili dall'agente all'interno dell'ambiente. Mediante l'attributo *observation space*, viene definito il numero di parametri che compongono lo stato, e per ognuno di essi il range di valori assumibili. L'attributo *reward range* contiene il minimo e massimo reward ottenibili nell' ambiente, di default settato a $(-\infty, +\infty)$.

Utilizzando la classe *Env* proposta dal framework come base per i nuovi ambienti, si adotta l'interfaccia comune prevista dal toolkit. In questo modo gli ambienti realizzati potranno essere integrati nella libreria del toolkit e le loro dinamiche potranno essere apprese da algoritmi già realizzati dagli utenti della community di openAI Gym.

3.2.2 Box2D

Box2D [9] è una libreria di simulazione 2D di corpi solidi realizzata per gestire il motore fisico nei videogames e in semplici simulazioni. Gli sviluppatori, possono utilizzarlo nei loro sistemi per muovere gli oggetti all'interno della scena in modo realistico rendendo l'ambiente di simulazione più interattivo. Dal punto di vista del game engine, il motore fisico non è altro che un sistema per la gestione delle animazioni procedurali ovvero generazione automatica delle animazioni in tempo reale. Box2D è scritto in portable C++, la versione scritta nel linguaggio Python, utilizzata per realizzare robotArm, prende il nome di pyBox2D [10]. Alla base di Box2D sono presenti diversi concetti e classi fondamentali; nel seguito del paragrafo sono introdotte le principali classi fornite dal motore grafico utilizzate per realizzare gli ambienti robotArm.

La classe principale definita da Box2D prende il nome di *World* la quale contiene tutti gli elementi che compongono l'ambiente e permette di gestire tutti gli aspetti della simulazione. La maggior parte delle interazioni con Box2D avvengono tramite la classe World, questo perché la simulazione, avviene mediante il metodo Step della classe World. È opportuno specificare il time step, la velocità e il numero di iterazioni del solver fisico ogni volta che si richiama il metodo Step sulla classe World. Dopo l'esecuzione del metodo è possibile osservare ed esaminare i componenti dell'ambiente i quali, in base all'azione eseguita dall'agente e le forze in gioco nell'ambiente fisico, assumeranno variazioni in quanto a posizione e rotazione. Le informazioni estratte saranno utili per il rendering della scena, per il calcolo del nuovo stato e del reward ottenuto oltre a determinare se l'obiettivo è stato raggiunto.

Per definire gli elementi che compongono l'ambiente, Box2D mette a disposizione le classi StaticBody e DynamicBody. Con la classe *StaticBody* è possibile definire un corpo statico all'interno di World. I corpi statici non entrano in collisione tra di essi e non possono essere mossi dal motore fisico, questo perché non sono soggetti a nessun tipo di forza. Con la classe *DynamicBody* si identifica un corpo il cui comportamento è completamente simulato dal motore grafico. Un corpo dinamico può entrare in collisione sia con altri corpi dinamici che con corpi statici.

Ogni corpo è definito da una relativa classe *BodyDef* richiesta nel costruttore di Body, la quale contiene informazioni fisiche come velocità, massa, posizione iniziale e angolo di rotazione iniziale. Oltre alle informazioni fisiche, bodyDef contiene la lista di forme che compongono il corpo definite mediante la classe *Shape* la quale è in grado di rappresentare forme come cerchi e poligoni. Per problemi derivanti al calcolo delle collisioni, Box2D permette la sola definizione di poligoni convessi. Per collegare un oggetto shape ad un body, occorre utilizzare la classe *Fixture* la quale aggiunge ulteriori informazioni alla forma come indice di restituzione, indice di attrito e densità. Per ogni fixture è inoltre possibile definire una categoria e una maschera, le quali saranno utilizzate dal motore fisico per eseguire filtri sulle collisioni tra i corpi.

Un ulteriore importante concetto definito da Box2D prende il nome di *joint*. I joint sono utilizzati per aggiungere vincoli di movimento ai body, con il mondo o tra di essi e possono essere combinati tra loro per creare interessanti movimenti. In alcune tipologie i joint sono provvisti di limiti per controllare il range dei movimenti e motori per guidare il joint ad una certa velocità. Tra la vasta gamma di tipologie

di joint definite da Box2D, in robotArm viene utilizzato il *Revolute Joint* il quale forza due corpi a condividere un punto in comune, anche chiamato *punto di cerniera.* Il Revolute Joint ha un singolo grado di libertà: la rotazione relativa tra i due corpi, chiamata joint angle. Per definire un RevoluteJoin è opportuno fornire al costruttore due corpi, e due punti ancora, uno per corpo, i quali saranno vincolati a rimanere sovrapposti.

3.2.3 Realizzazione ambienti robotArm

Grazie alla combinazione delle due tecnologie software sopracitate, è stato possibile realizzare gli ambienti in seguito alla loro progettazione. Gli ambienti robotArm implementano i metodi previsti da OpenAI Gym e mediante Box2D definiscono i corpi e i vincoli di movimento tra essi individuati in progettazione. La definizione e inizializzazione dei corpi avviene all'interno del metodo *reset* dell'ambiente, mentre la simulazione e i calcoli avvengono nel metodo *step*. Al termine di ogni step, invocando il metodo *render* dell'ambiente è possibile osservare l'avanzamento della simulazione. Il codice completo di robotArm è consultabile in appendice, nel seguito del paragrafo vengono commentate alcune sue parti, come la definizione di un segmento del braccio meccanico e i relativi vincoli di movimento, il calcolo del nuovo stato e del reward e l'utilizzo dei contactListener di Box2D per determinare quando l'oggetto è stato afferrato dalla pinza.

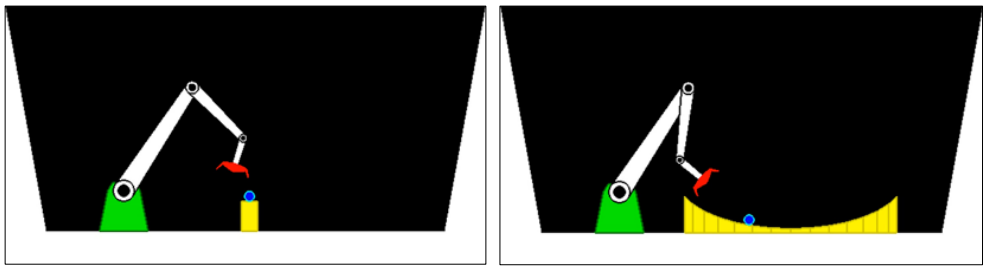

Figura 9 - a sinistra: robotArm Simple con supporto a piedistallo. a destra: robotArm Pipe con supporto ad half pipe

Nel metodo *reset* dell'ambiente, avviene la definizione e costruzione di tutti gli elementi che lo compongono: braccio meccanico, supporto all'oggetto (piedistallo, half pipe) e oggetto da afferrare. La definizione degli elementi dell'ambiente

avviene mediante l'utilizzo delle classi definite nel paragrafo 3.2.2. Nel seguito, mediante richiami al codice, viene mostrato come è stato possibile realizzare il supporto al braccio robotico, il primo segmento del braccio e i vincoli di movimento tra i due corpi.

```
self.shoulder = self.world.CreateStaticBody(
        position=(W/4,H/4),
        angle=0.0,
        fixtures=fixtureDef(
                shape=polygonShape(vertices=[(x/SCALE,y/SCALE) for x,y in
                        SHOULDER_POLY ]),
                density=0.2,
                friction=0.1,
                userData = True))
self.shoulder.color1 = (0,1,0)
self.shoulder.color2 = (0,0.5,0)
```

Figura 10 - Definizione del supporto statico per il braccio meccanico mediante le classi definite da Box2D

La base di appoggio del braccio meccanico è definita come corpo statico, questo perché si vuole rendere la sua posizione invariabile. La creazione di un corpo statico avviene mediante l'invocazione dell'oggetto world e richiede come parametri d'ingresso, gli attributi previsti dalla classe BodyDef, ovvero la posizione, l'angolo di rotazione iniziale del corpo e la lista di fixture che lo compongono. Osservando il codice in Figura 10, è possibile notare la definizione di shape all'interno di fixtureDef. In questo caso la forma del corpo statico è definita mediante una singola fixture la quale al suo interno, oltre ad informazioni quali densità e attrito, definisce una forma poligonale utilizzando come vertici i punti bidimensionali contenuti nel vettore SHOULDER_POLY. Negli ambienti robotArm, l'attributo userData di ogni fixture, mantiene un booleano, il quale definisce se in fase di rendering la fixture deve essere renderizzata o meno. Infine al corpo vengono settati due attributi colore, uno per l'area e uno per il perimetro.

```
self.armA = self.world.CreateDynamicBody(
        position = (W/4,H/3),
        angle=(90)*DEGTORAD,
        fixtures = fixtureDef(
                shape=polygonShape(vertices=[ (x/SCALE,y/SCALE) for x,y
                                        in ARM_A_POLY ]),
                density=0.2,
                friction=0.1,
                categoryBits=ARM_CAT,
                maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                restitution=0.0,
                userData = True)
                )
self.armA.color1 = (1,1,1)
self.armA.color2 = (0,0,0)

self.armA.CreateFixture(
        fixtureDef(
                shape=circleShape(radius=ARM_A_WIDTH/SCALE,
                        pos=((-ARM_A_LENGTH+7)/2/SCALE,0)),
                categoryBits=ARM_CAT,
                maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                userData = True))
```

Figura 11 - Definizione del corpo dinamico ArmA, rappresentante il primo segmento dei tre che compongono il braccio meccanico

Ogni segmento del braccio meccanico è definito come corpo dinamico e la sua forma è ottenuta combinando due forme: la prima è un poligono che ne definisce le dimensioni, mentre la seconda definisce una forma circolare con sole finalità estetiche posizionata alla base della prima forma. Le fixture dei segmenti del braccio robotico, sono munite di maschera e categoria per evitare collisioni non desiderate. In questo caso ad armA viene assegnata la categoria ARM, e come maschera vengono inserite le categorie con le quali si vogliono calcolare le collisioni, ovvero con il terreno, con l'oggetto da afferrare e con il piedistallo.

Definiti i due corpi è ora possibile vincolare i movimenti tra di essi mediante la definizione di un joint. Si vuole creare un punto di cerniera tra la base del braccio meccanico e il primo segmento, limitando i due corpi a soli movimenti angolari.

```
shoulderJoint = revoluteJointDef(
        bodyA = self.shoulder,
        bodyB = self.armA,
        localAnchorA=(0/SCALE,50/SCALE),
        localAnchorB=((-ARM_A_LENGTH+7)/2/SCALE,0/SCALE),
        enableMotor=True,
        enableLimit=True,
        maxMotorTorque=400.0,
        motorSpeed=0)

shoulderJoint.lowerAngle = 30*DEGTORAD-self.armA.angle
shoulderJoint.upperAngle = 90*DEGTORAD-self.armA.angle
self.shoulder.joint = self.world.CreateJoint(shoulderJoint)
```

Figura 12 - Definizione di revoluteJoint tra la base del braccio meccanico e il primo segmento

Osservando il codice in Figura 12, si nota che per definire un *revolute joint* è opportuno indicare i due corpi soggetti ai vincoli, e per ognuno di essi il punto di ancora. Il joint ha il compito di mantenere sovrapposti i due punti ancora permettendo solo variazioni angolari tra i corpi. Al joint definito, viene abilitato un motore, il quale è inizializzato a velocità pari a 0 con massima coppia a 400. La variazione della velocità del motore, consentirà all'agente di modificare il posizionamento del braccio durante gli step di simulazione. Il joint, inoltre, impone che l'angolo di rotazione tra i due corpi sia compreso tra 30° e 90°.

All'interno del metodo *step* dell'ambiente, in funzione dell'azione passata come parametro, vengono modificate le velocità dei motori dei diversi joint definiti nel metodo reset, e inseguito, mediante l'oggetto world, viene avanzata la simulazione di uno step. Al termine della simulazione è possibile osservare i nuovi valori dei corpi quali posizione, angolo di rotazione, velocità, e di conseguenza, calcolare il nuovo stato dell'ambiente e il reward ottenuto dall'agente.

```
deltaX,deltaY = self._getDeltaXY()
state = [
        self.clawOpened,
        self.shoulder.joint.angle,
        self.elbowjoint.angle,
        self.wristjoint.angle,
        deltaX,
        deltaY
        ]
```

Figura 13 - definizione dello stato dell'ambiente

Lo *stato* di entrambi gli ambienti, viene rappresentato mediante un vettore contenente le informazioni definite in fase progettuale in Tabella 1. I valori che compongono lo stato sono: stato della pinza del braccio robotico, gli angoli dei joint tra ogni coppia di segmenti del braccio e le distanze sugli assi x e y tra la pinza e l'oggetto da afferrare. In seguito all'esecuzione del metodo step sull'oggetto World, per ottenere i valori degli angoli dei joint basta invocare l'attributo *angle* sugli oggetti joint definiti nel metodo reset che legano i segmenti del braccio. Per quanto riguarda lo stato della pinza, viene memorizzato un valore booleano posto a true o false in base all'azione di apertura o chiusura eseguita dall'agente. Gli unici valori che richiedono di essere calcolati, sono le distanze tra pinza e oggetto, calcolate mediante il metodo getDeltaXY() il quale sottrae per ogni asse i valori tra il centro dell'oggetto e il centroide della pinza.

Un importante controllo eseguito nel metodo *step*, permette di determinare se l'obiettivo è stato raggiunto o meno. Per fare ciò occorre collegare un contact listener all'oggetto World di Box2d. In presenza di un contact listener, l'oggetto World, in fase di simulazione, richiama dei particolari metodi ogni qualvolta due fixture entrano o escono da una collisione. Grazie ai metodi predisposti dal contact listener, è possibile ottenere il numero di fixture della pinza in contatto con l'oggetto da afferrare. Come è possibile osservare dal codice in appendice, nel metodo reset viene definita la pinza, composta da due corpi rappresentanti le due branche che la compongono, entrambi ottenuti dall'unione di due fixture, questo perché le branche della pinza hanno forma concava, e dato che Box2d impedisce la definizione di shape concave, obbliga la costruzione di corpi concavi come combinazione di più shape convesse.

```
#ball captured checks
if self.clawDxCnt+self.clawSxCnt >= 3 and not self.captured:
   self.captured = not self.captured
elif self.clawDxCnt+self.clawSxCnt == 0 and self.captured:
   self.captured = not self.captured
```

Figura 14 – controllo del numero di fixture della pinza in contatto con l'oggetto, per determinare se l'obiettivo è stato raggiunto o meno.

Come si può osservare nel codice in Figura 14, l'ambiente determina che la pinza ha afferrato l'oggetto quando almeno tre sue fixture risultano contemporaneamente in contatto con l'oggetto. I contatori che compaiono nel codice sono incrementati e decrementati nei metodi del contact listener invocati dall'oggetto World, seguendo il design pattern observer.

```
self.reward = 0
if not self.done:
   if self.captured:
      self.reward = 200
      self.done = True
   elif self.ground_contact:
      self.done = True
   else:
      curDistance = round(self._getDistance(),2)
      if curDistance < self.prevDistance:
         self.reward = 1+(not self.clawOpened)*(-.2)
      else:
         self.reward = -1+(not self.clawOpened)*(-.2)
      if self.steps >= 500: self.done = True

self.prevDistance = round(self._getDistance(),2)
```

Figura 15 - Calcolo del reward in ambiente robotArm Simple all'interno del metodo step

Il codice in Figura 15, rappresenta il calcolo del reward in ambiente robotArm Simple. Per prima cosa si controlla se l'episodio è terminato o meno, questo perché, come previsto dal problema di reinforcement learning, in seguito al

completamento dell'episodio, l'agente non deve più ricevere reward. Il codice ricalca la funzione reward definita in fase di progettazione. Il Figura rappresenta il calcolo del reward per l'ambiente robotArm Simple nel quale, in caso di completamento dell'obiettivo l'agente ottiene un reward di 200, mentre in caso di reward intermedio, se l'azione eseguita riduce o incrementa la distanza tra pinza e oggetto l'agente riceve rispettivamente un reward pari a +1 o -1 con possibile variazione pari a 0.2 in base allo stato della pinza.

In fine si vuole mostrare come sono state combinate le due tecnologie Gym OpenAI e Box2D per simulare realisticamente gli ambienti e allo stesso tempo adottare un'interfaccia in grado di permettere l'interazione tra agente e ambiente prevista dal problema di reinforcement learning.

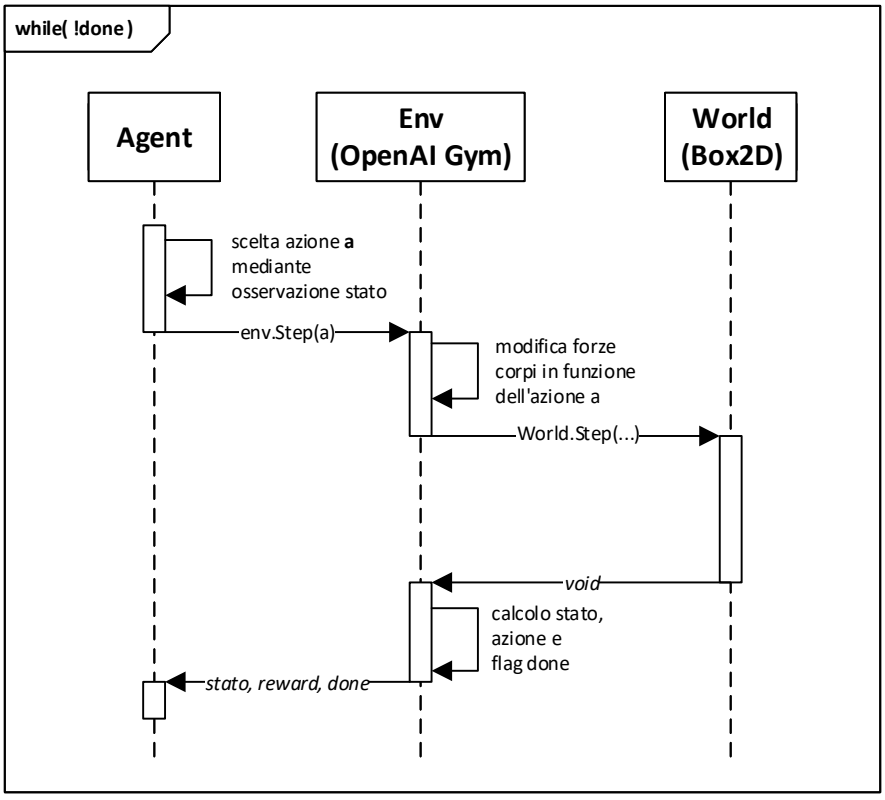

Figura 16 - diagramma di sequenza rappresentante l'interazione tra agente, ambiente (gym openAI) e World (box2D)

Come mostrato dal diagramma di sequenza in Figura 16, ad ogni interazione, l'agente sceglie l'azione da eseguire sulla base dello stato fornito dall'ambiente al

completamento dell'iterazione precedente, e invoca il metodo Step dell'ambiente, passando come parametro l'azione scelta. All'interno del metodo Step, l'ambiente modifica i valori dei corpi, definiti mediante box2d, in funzione dell'azione selezionata dall'agente e invoca il metodo Step di World il quale avanza la simulazione calcolando per ogni corpo le nuove velocità, posizioni e angolo di rotazione seguendo le regole fisiche e i vincoli imposti, e individuando le collisioni tra i corpi invocando di conseguenza i metodi del contact listener. Al termine della simulazione, la computazione torna al metodo Step dell'ambiente che, mediante l'osservazione dei nuovi valori dei corpi, calcolati dal metodo Step di World, è ora in grado di calcolare il nuovo stato, il reward e definire se l'episodio è terminato. Ultimati i calcoli, l'ambiente restituisce lo stato, il reward e il flag done all'agente il quale osservando il flag done per determinare se uscire dal ciclo o meno. Se si verifica il secondo caso, l'agente in base ai dati restituiti dall'ambiente, dovrà scegliere una nuova azione da eseguire sull'ambiente eseguendo un'ulteriore interazione con esso.

4 Implementazione Agenti

Nel seguito del capitolo vengono presentate e descritte le implementazioni dei metodi Q-Learning e Deep Q-Learning, per l'apprendimento delle dinamiche degli ambienti robotArm. Per ogni agente vengono descritte le scelte implementative e definite le classi e i metodi mediante richiami al codice. Per quanto riguarda il primo metodo, Q-Learning con materializzazione della Tabella Q, è stato necessario discretizzare i valori dai parametri dello stato degli ambienti e ridurne il numero, in quanto il metodo semplice prevede che per ogni possibile coppia stato-azione sia allocata una cella all'interno della tabella Q, nella quale sarà, in fase di apprendimento, memorizzata la stima del relativo valore. Discretizzando e riducendo il numero di parametri che rappresentano lo stato dell'ambiente, si perdono informazioni e di conseguenza gli stati perdono la proprietà di Markov in quanto gli stessi parametri di stato risultano in configurazioni dell'ambiente diverse. A causa della discretizzazione e riduzione degli input, il primo metodo realizzato non sarà in grado di raggiungere risultati eccellenti, per questo motivo è stato implementato un secondo metodo più avanzato, Deep Q-Learning, il quale sostituisce la tabella Q con una rete neurale volta all'approssimazione della funzione valore. Come già definito dettagliatamente (vedi sezione 2.3), grazie all'utilizzo della rete neurale, non avviene una vera e propria materializzazione del valore di ogni possibile coppia stato-azione, bensì si cerca di far stimare il comportamento della funzione valore alla rete neurale mediante la modifica dei suoi pesi interni. La rete neurale riceve in input valori continui e non richiede nessuna discretizzazione.

Anche se gli agenti implementano due algoritmi differenti, la modalità di interazione con l'ambiante rimane comune.

```
from robotic_arm_simple import RoboticArmSimple
from agent import Agent

nEpisodes = 1000
currentEpisode = 0

env = RoboticArmSimple()
agent = Agent(config)

while currentEpisode < nEpisodes:
    state1 = env.reset()
    done = False
    while not done:
        selectedAction = agent.ChooseAction(state1)
        state2, reward, done = env.step(selectedAction)
        agent.learn(state1,selectedAction,reward,state2)
        env.render()
        state1=state2
    currentEpisode+=1
```

Figura 17 - Codice Python per la interazione tra agente e ambiente

Il codice in Figura 17 mostra il metodo main nel quale sono istanziati i due oggetti: ambiente e agente. In seguito, il metodo presenta un doppio ciclo: Il ciclo più esterno itera gli episodi, e termina al raggiungimento dell'episodio memorizzato nella variabile *nEpisodes*. Ad ogni inizio di un nuovo episodio, viene richiamato il metodo reset sull'ambiente, il quale resetta l'ambiente e ne restituisce lo stato iniziale. Il ciclo interno itera tra gli step dell'episodio e al suo interno avviene l'invocazione dei metodi dell'agente per la scelta delle azioni, l'apprendimento dei valori delle coppie stato-azione e l'avanzamento della simulazione dell'ambiente. Ad ogni step, viene fornito all'agente lo stato corrente dell'ambiente, mediante il quale l'agente sceglierà l'azione da eseguire. Ottenuta l'azione, viene invocato il metodo step dell'ambiente passando l'azione scelta come parametro. Come previsto dall'interfaccia fornita da Gym OpenAI, il risultato di step comprende il nuovo stato, il reward ottenuto, e il flag done. L'apprendimento mediante la stima

dei valori delle coppie stato-azione di Q-Learning, richiede lo stato precedente, l'azione eseguita, il reward ottenuto e il nuovo stato dell'ambiente (vedi pseudocodice in Figura 5) e avviene mediante l'invocazione del metodo learn dell'agente. Infine è possibile richiamare il metodo render sull'ambiente per osservarne la simulazione. Un episodio termina quando, in seguito all'esecuzione di uno step di simulazione, il flag done assume valore true.

4.1 Implementazione Q-Learning

Q-Learning è un metodo Temporal Difference off-policy per il reinforcement learning. Come già definito dettagliatamente (vedi sezione 2.2), Q-Learning esegue bootstrapping, stimando i nuovi valori sulla base di stime passate, e non richiede un modello accurato dell'ambiente per apprenderne le dinamiche. Q-Learning è un metodo off-policy in quanto, per la stima dei valori delle coppie stato-azione, segue una policy completamente greedy, mentre per la scelta delle azioni ad ogni step, utilizza la policy ϵ-greedy in modo da consentire l'esplorazione di nuove coppie stato-azione.

Il primo agente realizzato implementa Q-Learning in versione base, materializzando la tabella delle coppie stato-azione. Come per l'ambiente, l'intera implementazione è in linguaggio Python. Il comportamento dell'agente risiede nella classe QLearnGreedy, nella quale è definito sia il metodo di scelta delle azioni, guidata dalla politica ϵ-greedy, con ϵ decrementata di una costante ad ogni episodio, che le dinamiche di apprendimento e stima dei valori delle coppie stato-azione. La classe è realizzata in modo da permettere la modifica dei parametri previsti dal metodo Q-Learning. Per inizializzare l'agente, oltre all'attributo action space dell'ambiente, contenente le possibili azioni eseguibili, occorre fornire in ingresso i parametri ϵ, α, γ che rappresentano rispettivamente l'indice di esplorazione, il tasso di apprendimento e l'indice di discounting. Dato che la politica di scelta delle azioni adottata è ϵ-greedy, con ϵ decrementata di una costante ad ogni episodio, la classe QLearnGreedy richiede un quarto parametro in ingresso: $\epsilon Step$ utilizzato nel calcolo di ϵ. Più formalmente l'indice di esplorazione viene calcolato come:

$$eps = \epsilon - \epsilon Step * episodio \tag{18}$$

Con variabile $episodio$ inizializzata a 0 e incrementata ad ogni inizio episodio, e valore di ϵ mantenuto costante per l'intera computazione.

Un'importante scelta implementativa è l'utilizzo di un dizionario al posto di una matrice per materializzare la tabella Q. La struttura dati dizionario [11], è build-in nel linguaggio Python. Utilizzando un dizionario al posto di una matrice, non risulta necessario inizializzare l'intera tabella comprensiva di tutte le possibili coppie stato-azione, evitando sia il problema relativo all'occupazione di spazio in memoria non utilizzato (caso in cui uno stato non viene mai visitato), che il calcolo del numero delle effettive coppie stato-azione possibili per l'inizializzazione della matrice. Inoltre grazie alla dotazione di un dizionario l'algoritmo di apprendimento è applicabile, senza modifiche, ad un diverso ambiente che prevede diversi parametri di stato, basandosi sul solo attributo action space, previsto dalla classe Env proposta dal toolkit Gym OpenAI (vedi sezione 3.2.1). Per quanto riguarda l'algoritmo di apprendimento, QLearnGreedy ricalca l'algoritmo Q-Learning (vedi pseudocodice in Figura 5) e prevede solo modifiche minori dovute all'utilizzo del dizionario al posto della tabella. Nel seguito mediante richiami al codice vengono descritti il metodo di scelta delle azioni e il metodo di apprendimento.

Il metodo *chooseAction* dell'agente richiede come parametri lo stato dell'ambiente, l'episodio corrente per il calcolo di ϵ e un valore booleano che, impostato a true fa restituire al metodo i valori delle coppie stato-azione oltre all'azione selezionata.

```
def getQ(self, state, action):
  return self.q.get((state, action), 0.1)

def getEps(self,episode):
  return max(0,self.epsilon-episode*self.epsStep)

def chooseAction(self, state, return_q=False,episode=0):
  q = [self.getQ(state, a) for a in self.actions]
  maxQ = max(q)
  if random.random() < self.getEps(episode):
    action = random.choice(self.actions)
    q=-1
  else:
    count = q.count(maxQ)
    if count > 1:
      best = [i for i in range(len(self.actions)) if q[i] == maxQ]
      i = random.choice(best)
    else:
      i = q.index(maxQ)

    action = self.actions[i]

  if return_q:
    return action, q
  return action
```

Figura 18 – Metodo chooseAction per la scelta dell'azione seguendo la policy ε-greedy nella classe QLearnGreedy

All'interno del metodo *ChooseAction* viene scelta l'azione in funzione dello stato corrente dell'ambiente. La scelta dell'azione segue la policy ϵ-greedy, la quale prevede in probabilità ϵ la scelta di un'azione casuale al posto dell'azione stimata con valore maggiore. Dopo aver estratto dal dizionario i valori relativi ad ogni azione eseguita sullo stato mediante il metodo getQ, il quale restituisce il valore se presente in dizionario altrimenti zero, viene memorizzato in una variabile separata, il valore più alto. In seguito viene generato un numero casuale il quale, se minore della probabilità ϵ calcolata mediante la formula (18) dal metodo *getEps*, determina la scelta casuale dell'azione. Al contrario, viene scelta l'azione stimata migliore. In

questo caso però è possibile che più azioni abbiano il medesimo valore, per questo motivo viene eseguito un controllo sul numero di azioni aventi il valore più alto e in seguito, viene scelta casualmente una delle azioni stimate migliori. Infine viene restituito l'indice dell'azione scelta e, se richiesto, il vettore dei valori q.

Il metodo *Learn* di QLearnGreedy richiede come parametri d'ingresso lo stato, l'azione eseguita, il reward e il nuovo stato raggiunto e ha l'obiettivo di stimare il valore di una coppia stato-azione seguendo la formula prevista dal metodo Q-Learning (15).

```
def learn(self, state1, action1, reward, state2):
   maxqnew = max([self.getQ(state2, a) for a in self.actions])
   self.learnQ(state1, action1, reward,
                value=reward + self.gamma*maxqnew)

def learnQ(self, state, action, reward, value):
      oldv = self.q.get((state, action), None)
      if oldv is None:
         self.q[(state, action)] = value
      else:
         self.q[(state, action)] = oldv + self.alpha * (value - oldv)
```

Figura 19 - Metodi learn e learnQ per la stima del valore stato-azione nella classe QLearnGreedy

L'apprendimento avviene mediante l'invocazione del metodo learn, il quale a sua volta, dopo aver ottenuto il valore più alto ottenibile nel nuovo stato visitato, invoca un secondo metodo learnQ passando come parametri lo stato, l'azione, il reward e il valore stimato per la coppia stato-azione. Il metodo learnQ controlla se esiste già una stima del valore per la coppia stato-azione in dizionario. In caso negativo, inserisce in dizionario il nuovo valore, altrimenti aggiorna il valore seguendo la formula (15). È stato scelto di dividere l'apprendimento in due metodi separati, in quanto nel caso di stato terminale, il valore dello stato dipende dal solo reward ottenuto dato che si suppone che tutti gli stati seguenti restituiscano un valore reward pari a zero. Per questo motivo, in caso di stato terminale, vene invocato learnQ al posto di learn, passando come parametro value il reward ottenuto.

Come anticipato nell'introduzione al capitolo, l'apprendimento per Q-Learning con dizionario, viene eseguito con variabili di stato ridotte e discretizzate. I parametri di stato relativi agli angoli dei giunti: angolo spalla, angolo gomito e angolo polso (vedi Tabella 1) sono stati eliminati, mentre i parametri DeltaX e DeltaY sono stati discretizzati riducendo la precisione ad un solo decimale.

Indice	***Parametro***	***Range***	***Numerosità***
0	Stato Pinza	0 – 1	2
4	Delta X	-8 – 20	281
5	Delta Y	-13 – 13	261
		Numerosità totale:	146682

Tabella 3 - Rappresentazione dello stato degli ambienti, utilizzato in fase di apprendimento dall'agente QLearnGreedy. La colonna "numerosità" mostra il numero di possibili valori assumibili dal relativo parametro discretizzato

In Tabella 3 sono riportati i parametri di Tabella 1 utilizzati per rappresentare gli stati degli ambienti per l'agente QLearnGreedy. Oltre alle informazioni presenti in Tabella 1, in Tabella 3 viene mostrato il numero di valori assumibili da ogni parametro e il numero totale di possibili stati. Per Delta X e Delta Y, la numerosità riportata rappresenta il numero di valori assumibili in seguito alla discretizzazione. Si noti che il numero riportato, non equivale alla dimensione massima assumibile dal dizionario, in quanto nel dizionario sono memorizzate le stime dei valori per ogni coppia stato-azione, perciò occorre moltiplicare la numerosità totale per il numero di azioni eseguibili negli ambienti, ottenendo:

$$Dimensione\ Dizionario = 2 * 281 * 261 * 8 = \ 1173456$$

Considerando l'allocazione di 64 bit (float64) per ogni coppia stato-azione, il dizionario assume un peso massimo in memoria pari a 8.95 MB.

Per quanto riguarda i parametri richiesti dal metodo: ϵ, α, γ e $\epsilon Step$, la scelta è avvenuta in seguito a diversi test eseguiti variando i valori ϵ, $\epsilon Step$ e γ (vedi Tabella 4). I parametri ϵ e $\epsilon Step$ determinano il numero di episodi richiesti dall'apprendimento, in quanto raggiunto $\epsilon = 0.1$ l'agente esegue una minima

esplorazione, stabilizzando i risultati, in quanto ad ogni step la politica ϵ-greedy di scelta delle azioni sceglierà con maggiore probabilità l'azione avente valore più alto. I due parametri sono stati scelti in modo da ridurre a 0.1 l'indice di esplorazione dopo l'esecuzione di 125000 episodi. Per quanto riguarda il parametro γ, in letteratura, per ambienti di natura episodica si propone di mantenere $\gamma = 0.99$, in quanto il limite al discounting dei valori avviene automaticamente grazie al reset dell'ambiente al raggiungimento di uno stato terminale. In questo caso però è stato constatato che con $\gamma = 0.6$ è possibile ottenere prestazioni migliori, questo per via della perdita della proprietà di Markov in seguito alla riduzione e discretizzazione dei parametri di stato. Dando meno peso ai valori ottenibili nel futuro, mediante la riduzione del parametro γ, è stato possibile ottenere prestazioni migliori dall'agente QLearnGreedy.

Parametro	***Valore***	***Descrizione***
ϵ	0.6	Indice di esplorazione iniziale. Rappresenta la probabilità di scelta di un'azione casuale al posto dell'azione stimata migliore
$\epsilon Step$	$4 * 10^{-6}$	Decremento di ϵ ad ogni inizio di un nuovo episodio, per i valori scelti $\epsilon = 0.1$ dopo il termine di 125000 episodi
γ	0.6	Indice di discounting. Rappresenta l'apporto dato dai valori degli stati successivi, nel calcolo del valore di una coppia stato-azione
α	0.2	Fattore di apprendimento. Rappresenta l'indice di aggiornamento del valore di una coppia stato-azione

Tabella 4 - Riepilogo dei parametri scelti per Q-Learning con tabella materializzata

4.2 Implementazione Deep Q-Learning

Data la perdita della proprietà di Markov sugli stati dell'ambiente nel primo agente, dovuta alla diminuzione e discretizzazione dei parametri, non è possibile ottenere risultati eccellenti. Purtroppo la discretizzazione è inevitabile, in quanto in sua assenza, l'elevata richiesta di memoria per la memorizzazione del dizionario

sarebbe proibitiva, ma soprattutto, il numero di dati e tempo richiesti per stimare accuratamente ogni singola coppia stato-azione sarebbero troppo elevati. La soluzione al problema è combinare il metodo di reinforcement learning, Q-Learning ad un metodo di approssimazione di funzione. Si è scelto di implementare il metodo Deep Q-Learning con experience replay (vedi sezione 2.3). Nel metodo Deep Q-Learning viene sostituita la tabella Q con una rete neurale volta all'approssimazione della funzione valore. L'adozione di una rete neurale permette di utilizzare lo stato completo fornito dagli ambienti senza riduzioni e discretizzazioni.

Per la scelta delle azioni, Deep Q-Learning adotta una politica ϵ-greedy, simile alla politica utilizzata dal primo agente. In questo caso però, i valori delle coppie stato-azione vengono ottenuti eseguendo un passo forward sulla rete neurale. Il metodo implementato, in fase di apprendimento, stima i valori target per il calcolo della loss function (16), seguendo la logica proposta dai metodi Monte Carlo (vedi sezione 2.1.3) la quale prevede la stima dell'expected return di una coppia stato-azione, mediante le informazioni memorizzate in replay memory. In Deep Q-Learning, ad ogni interazione, l'agente memorizza una nuova tupla in replay memory contenente: lo stato precedente, l'azione eseguita, il reward ottenuto, il nuovo stato visitato, il flag done, il valore, e il puntatore alla tupla successiva.

Indice	***Parametro***	***Valore***	***Descrizione***
0	stato1	Float[6]	Stato precedente dell'ambiente
1	azione	Int	Indice dell'azione eseguita
2	reward	Float	Reward ottenuto
3	stato2	Float[6]	Stato seguente, raggiunto con l'esecuzione dell'azione sullo stato1
4	notDone	Boolean	Valore negato del flag done, il quale determina se lo stato è terminale o meno
5	qValue	Float	Valore della coppia stato-azione
6	puntatore	Int	Puntatore alla tupla seguente

Tabella 5 - Valori contenuti per ogni tupla in replay memory

In questo modo scegliendo casualmente una tupla in replay memory, sfruttando il puntatore alla tupla successiva, è possibile stimarne l'expected return (10). In seguito alla stima, l'expected return calcolato viene salvato in posizione 5 nella relativa tupla. Se la tupla selezionata dalla replay memory appartiene ad un episodio non ancora terminato, il calcolo dell'expected return viene totalmente o in parte calcolato sfruttando le stime della rete neurale.

Al posto di eseguire il training alla rete in base alla sola transizione appena eseguita, come avviene nel primo metodo implementato, in Deep Q-Learning, il training avviene sulla base di un sotto insieme di transizioni selezionate in modo casuale dalla replay memory. Questa tipologia di aggiornamento prende il nome di aggiornamento a mini-batch e porta un notevole vantaggio rispetto al metodo base, in quanto in fase di apprendimento saranno richiesti meno episodi per convergere a una soluzione ottimale. Per realizzare la rete neurale è stata utilizzata la libreria software open source TensorFlow [12] la quale permette di modellare in poche righe di codice una rete neurale. TensorFlow implementa diversi algoritmi di ottimizzazione per l'apprendimento mediante discesa del gradiente, tra cui RMSProp [13], ottimizzazione per l'apprendimento a mini-batch utilizzato dagli autori del metodo Deep Q-Learning [6]. Per inizializzare l'agente Deep Q-Learning, oltre all'attributo action space dell'ambiente, contenente le possibili azioni eseguibili, e i parametri $\epsilon, \epsilon Step, \alpha$ e γ i quali rispettivamente rappresentano, l'indice di esplorazione, il fattore di decremento dell'indice di esplorazione, il tasso di apprendimento e l'indice di discounting, occorre fornire ulteriori parametri quali: dimensione massima della replay memory (mem_size), dimensione dei mini-batch ($batch\_size$) e probabilità di aggiornamento ($prob\_update$). Nel seguito vengono descritti i metodi di apprendimento e scelta delle azioni.

Il metodo *ChooseAction* di Deep Q-Learning implementa la politica ϵ-greedy analogamente al metodo di scelta delle azioni discusso precedentemente per Q-Learning base. In questo caso però, le stime dei valori sono ottenute mediante la rete neurale.

```
def ChooseAction(self, state, episode=None):
    eps = self.epsilon(episode)
    # epsilon greedy
    if np.random.random() > eps:
        action = self.argmaxq(state)
    else:
        action = self.action_space.sample()
    return action

def argmaxq(self, state):
    return np.argmax(
            self.sess.run(self.Q, feed_dict={self.x: state}))
```

Figura 20 - Metodo ChooseAction di Deep Q-Learning e metodo argmaxq per il calcolo dei valori mediante passo forward sulla rete Q

Si osservi nel codice in Figura 20, che in caso di scelta dell'azione con expected return maggiore, il metodo ChooseAction invoca un secondo metodo *argmaxq* richiedente come parametro d'ingresso lo stato dell'ambiente, utilizzato internamente come input alla rete neurale Q, per ottenere i valori relativi ad ogni possibile azione eseguibile. Nel metodo *argmaxq* in Figura 20, l'oggetto sess rappresenta la sessione corrente in tensorflow. Eseguendo il metodo run sull'oggetto sessione, è possibile interagire con la rete neurale modellata, in questo caso viene eseguito un passo forward con stato dell'ambiente come input. L'output ottenuto risulta in un vettore, di lunghezza pari al numero di azioni eseguibili. Ottenuto il vettore, il metodo restituisce l'indice dell'azione con valore maggiore.

L'apprendimento della funzione valore avviene mediante l'invocazione del metodo *learn,* il quale richiede in ingresso: lo stato, l'azione scelta, il nuovo stato ottenuto, il reward ottenuto, e il flag done negato. Il flag done viene richiesto negato (valore del flag pari a 0 se lo stato è terminale, 1 altrimenti) in quanto nel calcolo del valore target viene utilizzato il valore negato del flag done, in modo da annullare il reward in caso di stato terminale, come previsto dal reinforcement learning. Il metodo esegue sia l'inserimento della nuova tupla che la scelta casuale di un sottoinsieme di tuple dalla replay memory. In seguito per ogni tupla selezionata viene calcolato il valore target in modo da determinare l'errore degli

output della rete da minimizzare mediante l'aggiustamento dei pesi. L'implementazione prevede che in fase di apprendimento, la coppia stato-azione passata in input al metodo, sia compresa nel mini-batch per il calcolo dell'aggiornamento, inoltre per ridurre il costo computazionale complessivo, l'aggiornamento alla rete viene eseguito con probabilità pari al parametro $prob\_update$, ciò significa che con probabilità pari a $1 - prob\_update$, la tupla passata come parametro al metodo *learn* viene inserita in replay memory, ma non viene eseguito l'aggiornamento a mini-batch.

```
#calcolo valore target della coppia stato-azione passata come parametro
target = reward + self. gamma * self.maxq(newstate) * notdone
target = target.reshape(1, )

#inserimento del nuovo stato nel mini-batch
allstate = state.reshape(1, -1)
allaction = np.array([action])
alltarget = target
```

Figura 21 - frammento di codice del metodo learn di Deep Q-Learning. Calcolo valore target della coppia stato-azione passata come parametro, e inserimento della coppia stato-azione nel mini-batch

Il codice in Figura 21 mostra il calcolo del valore target della coppia stato-azione passata come parametro al metodo. In questo caso il calcolo del valore target viene stimato dalla rete neurale in quanto non ancora presente in replay memory. In seguito, informazioni quali stato, azione e valore target sono inserite nei rispettivi vettori, ai quali, in caso di aggiornamento, saranno aggiunti i valori relativi alle tuple selezionate casualmente dalla replay memory.

```
#selezione casuale delle tuple dalla replay memory
ind = np.random.choice(len(self.memory), self.config['batch_size'])

#per ogni tupla selezionata
for j in ind:
    state1, action, reward, state2, notDone, qValue, nextTuple = self.memory[j]
    #se qValue non è nullo, non viene eseguita la stima
    if qValue != None:
        target = qValue
    else:
        #stima dell'expected return
        target = reward
        gamma = self.gamma
        offset = 0
        if nextTuple == None:
            target += gamma * self.maxq(state2) * notDone

        while nextTuple != None:
            offset += nextTuple
            n = j + offset
            target += gamma * self.memory[n][2]
            #decremento esponenziale di gamma
            gamma = gamma * self.gamma
            if self.memory[n][6] == None:
                target +=
                    gamma * self.maxq(self.memory[n][3]) * self.memory[n][4]
            nextTuple = self.memory[n][6]

        self.memory[j][5] = target

    currentValue = r + self.gamma * self.maxq(state2) * notDone
    target =
        target * self.alpha + (currentValue) * (1. - self.alpha)

    alltarget = np.concatenate((alltarget, target))
    allstate = np.concatenate((allstate, state1.reshape(1, -1)))
    allaction = np.concatenate((allaction, np.array([action])))
```

Figura 22 - Frammento di codice del metodo learn di Deep Q-Learning. In seguito alla selezione casuale delle tuple in replay memory, stima dei relativi expected reward mediante metodo Monte Carlo combinato alle stime dei valori date dalla rete neurale e inserimento dei valori nei relativi vettori.

In caso di aggiornamento, per ogni tupla selezionata casualmente dalla replay memory, si ottiene il valore target. Se la tupla viene selezionata per la prima volta, occorre stimarne il valore target, in caso contrario la stima del valore è presente

all'interno della tupla. Osservando il codice in Figura 22 si nota che il calcolo avviene in modo iterativo, navigando tra gli stati seguenti mediante il puntatore *nextTuple*. Se la tupla selezionata è parte di un episodio portato a termine, l'intera stima avviene seguendo il principio di stima dei metodi Monte Carlo, in caso contrario, l'ultimo valore sommato alla variabile target, è stimato mediante la rete neurale. Al termine, la stima viene salvata all'interno della tupla. Infine il valore stimato viene combinato al valore corrente in output alla rete, seguendo la formula di aggiornamento dei valori delle coppie stato-azione prevista da Q-Learning ottenendo il valore target finale. In seguito i valori di stato, azione e target delle tuple selezionate, vengono aggiunti ai relativi vettori, i quali al termine avranno dimensione pari a $batch\_size + 1$.

```
allactionsparse = np.zeros((allstate.shape[0], self.n_out))
allactionsparse[np.arange(allaction.shape[0]), allaction] = 1

self.sess.run(self.optimizer,
      feed_dict={self.x: allstate,
                 self.y: alltarget,
                 self.curraction: allactionsparse})
```

Figura 23 - frammento di codice del metodo learn di deep Q-Learning. Costruzione della matrice allactionsparse e esecuzione del passo backward sulla rete neurale.

In seguito alla costruzione della matrice contenente i vettori one-hot[3] di lunghezza pari al numero di azioni (8 per gli ambienti robotArm), con valore della cella relativa all'indice dell'azione posto a 1, viene eseguito l'aggiornamento dei pesi della rete mediante l'invocazione del metodo run sull'oggetto sessione di Tensorflow, fornendo come parametri: il vettori degli stati, il vettore dei valori target e la matrice contenente i vettori one-hot relativi alle azioni. Infine il metodo *learn,* sia in caso di aggiornamento che in caso contrario, inserisce la tupla fornita in ingresso, in coda alla replay memory, modificando se opportuno il puntatore nextTuple della tupla inserita in coda all'iterazione precedente, in modo da puntare alla nuova tupla inserita. Nel caso in cui l'inserimento della nuova tupla superi la dimensione

[3] Vettore one-hot: vettore contenente tutti i valori posti a 0 tranne uno ad 1.
Es. 00100000

massima della replay memory mem_size, viene mantenuta la dimensione massima eliminando le tuple in eccesso seguendo logica FIFO.

La rete neurale modellata, rappresentata graficamente in Figura 24, è composta da tre strati di cui uno hidden fully connected. Il primo strato di input è composto da 6 neuroni, uno per ogni parametro di stato fornito degli ambienti (vedi Tabella 1). Lo strato hidden è composto da 300 neuroni con attivazione a tangente iperbolica, mentre lo strato di output prevede un neurone per ogni possibile azione (vedi Tabella 2) per un totale di 8 con attivazione lineare, in quanto gli output dovranno stimare i valori delle coppie stato-azione. Sia i neuroni dello strato hidden che i neuroni dello strato output sono muniti di bias, inizializzato a 0. I valori assunti dai pesi della rete sono regolarizzati mediante L2loss per evitare overfitting, e sono inizializzati casualmente seguendo una distribuzione normale troncata.

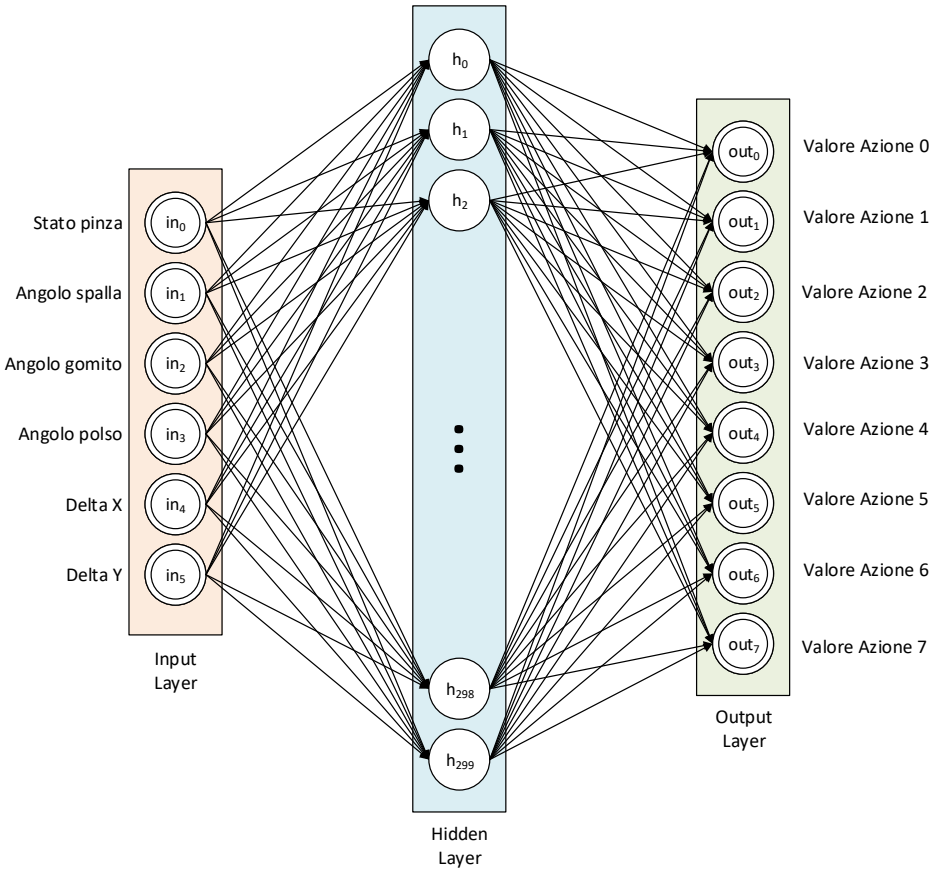

Figura 24 - Rappresentazione grafica della rete neurale adottata

L'aggiornamento dei pesi della rete avviene in funzione della minimizzazione dell'errore medio sul mini-batch degli output della rete, rispetto ai valori target stimati. Per ottimizzare l'aggiornamento dei pesi viene utilizzato l'algoritmo RMSProp [13], versione a mini-batch dell'algoritmo RProp implementato nativamente in TensorFlow. RMSProp ottimizza la discesa del gradiente mediante l'utilizzo della magnitudine dei recenti gradienti per normalizzare il gradiente, infatti RMS sta per Root Mean Squared. La scelta dell'algoritmo di ottimizzazione per il calcolo del gradiente è stata guidata dai dettagli implementativi forniti da DeepMind [6].

Parametro	***Valore***	***Descrizione***
Dimensione Input Layer	6	Numero dei neuroni che compongono il primo strato della rete neurale, numerosità equivalente al numero dei parametri previsti dallo stato degli ambienti
Dimensione hidden Layer	300	Numero dei neuroni che compongono il secondo strato della rete neurale, unico stato hidden della rete
Dimensione Output Layer	8	Numero dei neuroni che compongono il terzo ed ultimo strato della rete neurale, numerosità equivalente al numero di azioni degli ambienti
Regolarizzazione pesi 1	0.00001	Regolarizzazione applicata ai pesi tra lo strato input e lo strato hidden
Regolarizzazione pesi 2	0.0000001	Regolarizzazione applicata ai pesi tra lo strato hidden e lo strato output
Initial_learnrate	0.012	Tasso di apprendimento iniziale dato in input all'algoritmo RMSProp
Decay_learnrate	0.997	Fattore di decremento esponenziale del tasso di apprendimento iniziale *Initial_learnrate*
mementum	0.05	Parametro previsto dall'algoritmo RMSProp che agisce come filtro di irregolarità locali per evitare convergenza in minimi locali

Tabella 6 – Riepilogo dei parametri scelti per la rete neurale

Per quanto riguarda i parametri richiesti dal metodo Deep Q-Learning con replay memory, è stato scelto $\epsilon = 0.45$ con decay pari a $\epsilon Step = 0.993$, in questo caso l'indice di esplorazione decresce esponenzialmente raggiungendo valori prossimi allo zero intorno ai 1000 episodi. Dato l'utilizzo dell'intero stato degli ambienti in assenza di discretizzazione, il valore dell'indice di discounting scelto è $\gamma = 0.99$ come proposto dalla letteratura in caso di ambienti episodici, mentre il fattore di apprendimento per l'aggiornamento dei valori è pari ad $\alpha = 0.1$. La dimensione massima della replay memory è pari a $mem\_size = 150000$, ciò significa che nel caso peggiore, in replay memory sono contenuti 300 episodi, in quanto il numero massimo di step previsto dagli ambienti per un episodio è pari a 500. Gli aggiornamenti dei pesi della rete vengono eseguiti con probabilità $prob\_update = 0.25$ con mini-batch di dimensione $batch\_size = 75$.

Parametro	***Valore***	***Descrizione***
ϵ	0.45	Indice di esplorazione iniziale. Rappresenta la probabilità di scelta di un'azione casuale al posto dell'azione stimata migliore
$\epsilon Step$	0.993	Decremento esponenziale di ϵ ad ogni inizio di un nuovo episodio, per i valori scelti $\epsilon = 0$ dopo il termine di 1000 episodi
γ	0.99	Indice di discounting. Rappresenta l'apporto dato dai valori degli stati successivi, nel calcolo del valore di una coppia stato-azione
α	0.1	Fattore di apprendimento. Rappresenta l'indice di aggiornamento del valore di una coppia stato-azione
mem_size	150000	Dimensione massima della replay memory
$batch\_size$	75	Dimensione dei mini-batch per ogni aggiornamento dei pesi della rete
$prob\_update$	0.25	Probabilità di aggiornamento dei pesi della rete ad ogni interazione tra ambiente e agente

Tabella 7 – Riepilogo dei parametri scelti per Deep Q-Learning

5 Risultati Sperimentali

Nel seguito del capitolo vengono mostrati i risultati ottenuti dai metodi Q-Learning, e Deep Q-Learning sulle diverse configurazioni degli ambienti robotArm. Per ognuna delle quattro configurazioni di robotArm, sono state misurate le performance di apprendimento delle dinamiche dei due diversi agenti, con parametri in input invariati rispetto ai valori riportati nel capitolo precedente, osservabili nelle tabelle riepilogative Tabella 4 e Tabella 6. Per quanto riguarda le modalità di calcolo delle performance dei due agenti, è stato scelto di considerare sia le performance di accuratezza e total reward che il quantitativo di tempo richiesto per ottenere le performance finali. Per ogni test, accuratezza e total reward vengono riportati in due modalità: la prima riporta gli indici di performance più alti ottenuti in fase di training, calcolati sulla media dei valori ottenuti in 100 episodi consecutivi, la seconda riporta accuratezza e total reward ottenuti in media su 500 episodi al termine del training. Inoltre per ogni test eseguito, sono riportati due diversi grafici: Il primo mostra l'andamento dei total reward medi in funzione del numero di episodi terminati, mentre il secondo mostra la percentuale di episodi completati correttamente in funzione del numero di episodi portati a termine, dove per episodio completato si intende un episodio in cui l'agente ha afferrato l'oggetto con successo entro i 500 step. Il primo grafico proposto per l'osservazione dell'andamento delle performance è stato scelto seguendo le dinamiche di stima utilizzate da Gym OpenAI per confrontare e validare i risultati degli algoritmi pubblicati dagli utenti della community, e permette di verificare se l'agente è in grado di stimare correttamente la funzione valore ottima, massimizzando i total reward. Mediante il secondo grafico, invece, è possibile verificare se attraverso l'apprendimento per rinforzo un agente è in grado di apprendere autonomamente come afferrare un oggetto. In questo modo è possibile osservare l'accuratezza di completamento del task senza considerare l'ottenimento della funzione valore ottima, osservabile mediante il primo grafico. Data l'elevata differenza del numero di episodi richiesti dai due agenti per convergere ad una soluzione, i grafici presentano valori medi in diverse granularità: per l'agente Q-Learning, i grafici presentano i valori medi calcolati sui valori di 100 e 5000 episodi, in quanto il numero totale di episodi richiesti dal training è pari a 150000. Per quanto riguarda invece il secondo agente, Deep Q-Learning, i grafici che illustrano l'andamento del total reward, presentano i valori ottenuti per ogni singolo episodio, e i valori

calcolati in media su 100 episodi, in quanto il numero totale di episodi richiesti per il training è pari a 2000.

Nel seguito del capitolo, per ogni configurazione dell'ambiente realizzata, vengono riportati e commentati i grafici ottenuti dagli agenti Q-Learning e Deep Q-Learning. Infine vengono confrontati i due metodi, riepilogando i valori di performance ottenuti.

5.1 Risultati ottenuti in ambiente robotArm Simple

Con robotArm Simple si identifica l'ambiente robotArm con posizione dell'oggetto statica, variata casualmente ad ogni inizio episodio nei due assi X e Y. robotArm Simple presenta un piedistallo con posizione e altezza casuale sul quale viene posizionato l'oggetto. Nel seguito del paragrafo vengono riportati e commentati i grafici di performance degli agenti Q-Learning e Deep Q-Learning.

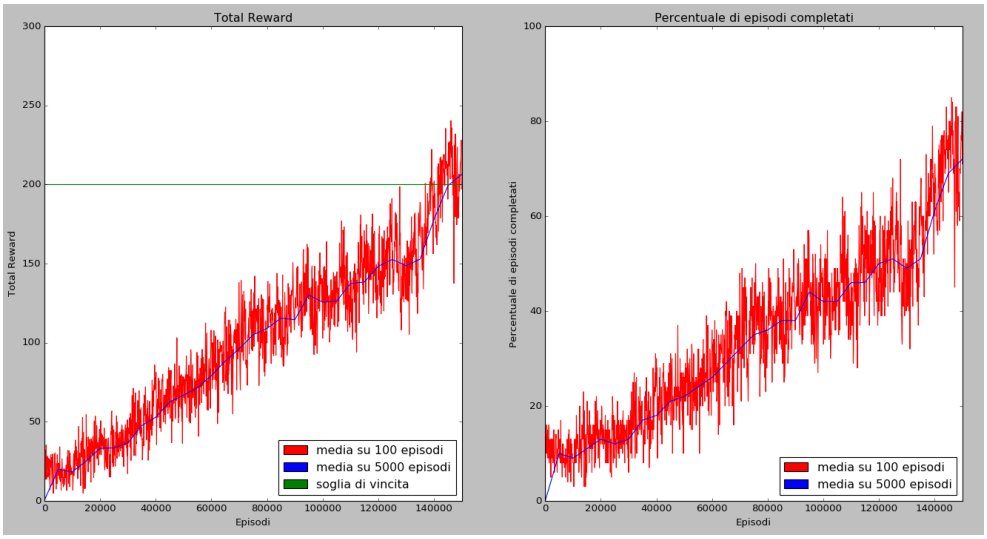

Figura 25 – Grafici delle performance per Q-Learning in robotArm Simple. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 25 mostrano le performance del metodo Q-Learning in fase di training in ambiente robotArm Simple. Come anticipato, per Q-Learning il numero di episodi richiesti in fase di training è pari a 150000, per questo motivo i grafici presentano valori aggregati ottenuti come media dei valori di 100 e 5000 episodi. In seguito a 125000 episodi, l'indice di esplorazione ϵ viene ridotto a 0.1

riducendo di conseguenza l'esplorazione e stabilizzando i risultati ottenuti. Anche se i grafici potrebbero sembrare troncati prematuramente, data la riduzione dell'indice di esplorazione, i valori ottenuti negli episodi seguenti si stabilizzano rispetto ai risultati ottenuti al termine degli episodi illustrati nei grafici.

Osservando il grafico di destra, riportante la percentuale di episodi completati con successo, si nota che al termine del training, l'agente completa con successo più del 75% degli episodi. Data la riduzione e discretizzazione dello stato dell'ambiente, non è stato possibile ottenere risultati migliori, in quanto diverse configurazioni dell'ambiente risultano in valori di stato equivalenti, introducendo un notevole errore nel calcolo dei valori delle coppie stato-azione. Il grafico di sinistra, riportante in media il valore dei reward totali ottenuti negli episodi, ricalca l'andamento del grafico di destra in quanto in robotArm Simple, il reward ottenuto al completamento dell'obiettivo è sempre pari a 200. Per questo motivo nella configurazione Simple di robotArm, il total reward medio è fortemente correlato alla percentuale di episodi completati correttamente mostrata nel grafico di destra.

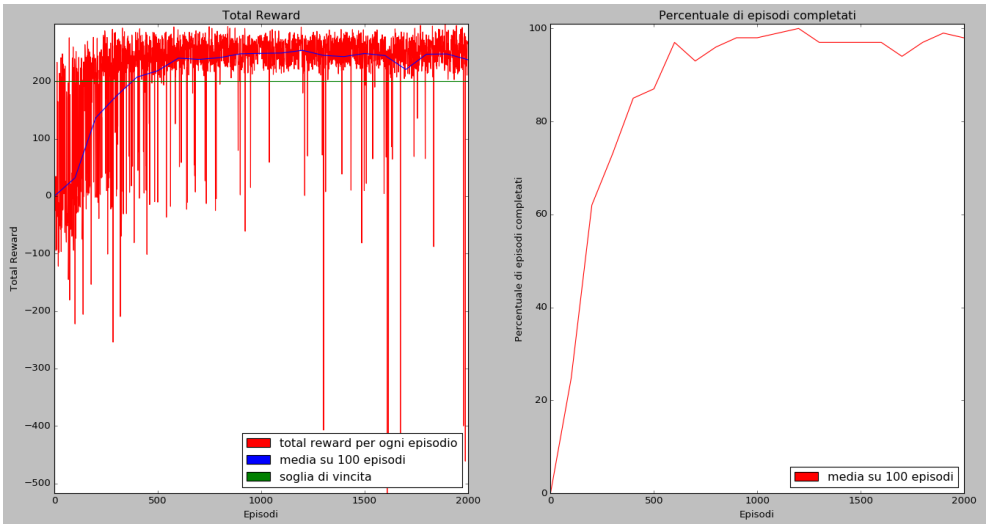

Figura 26 - Grafici delle performance per Deep Q-Learning in robotArm Simple. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 26 mostrano le performance del metodo Deep Q-Learning in fase di training in ambiente robotArm Simple. Come anticipato, il numero di episodi richiesti per il training dell'agente Deep Q-Learning è pari a 2000,

per questo motivo, per l'agente Deep Q-Learning, il grafico dell'andamento del total reward riporta i valori ottenuti in ogni singolo episodio e il valore medio ottenuto su 100 episodi. Osservando il grafico di destra, è possibile notare che Deep Q-Learning è in grado di completare con successo più del 95% degli episodi dopo il training su soli 500 episodi. Grazie alla replay memory e alla possibilità di utilizzare l'intero stato dell'ambiente senza l'obbligo di discretizzarne i valori, deep Q-Learning riesce ad ottenere risultati eccellenti in ambiente robotArm Simple. Anche in questo caso, il grafico di sinistra, data la natura della funzione reward di robotArm Simple, risulta ricalcare il grafico di destra.

5.2 Risultati ottenuti in ambiente robotArm Pipe

Con robotArm Pipe si identifica l'ambiente robotArm con posizione dell'oggetto in continuo movimento oscillatorio. Per ottenere le specifiche, robotArm Pipe presenta un half pipe sul quale l'oggetto da afferrare, è in continuo moto oscillatorio. Nella versione Pipe di robotArm, l'half pipe assume posizione e dimensione statica e sia la posizione iniziale che la velocità dell'oggetto oscillante sono equivalenti per ogni episodio. Nel seguito del paragrafo vengono riportati e commentati i grafici di performance degli agenti Q-Learning e Deep Q-Learning.

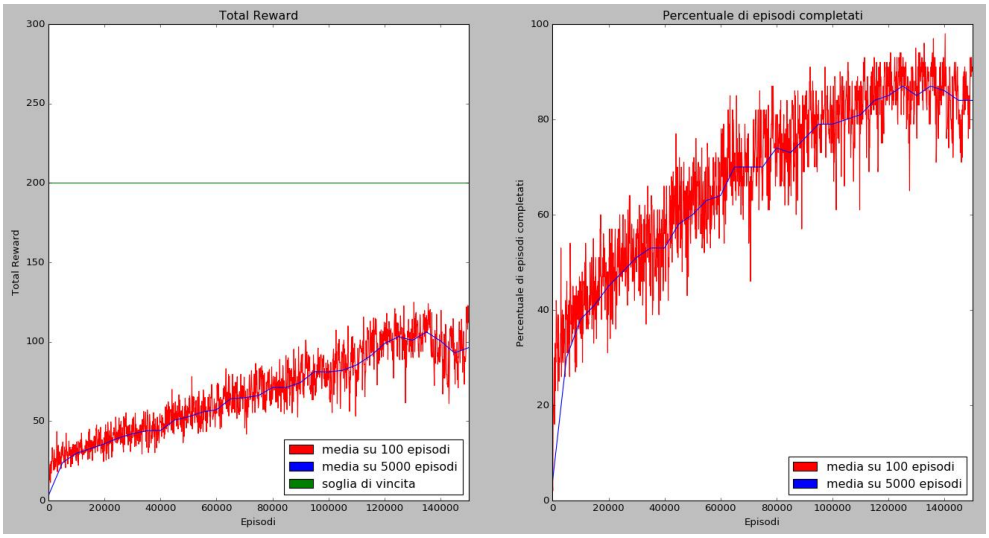

Figura 27 - Grafici delle performance per Q-Learning in robotArm Pipe. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 27, mostrano le performance del metodo Q-Learning in fase di training in ambiente robotArm Pipe. Osservando il grafico di destra, riportante la percentuale di episodi completati in funzione del numero di episodi portati a termine, si nota che al termine dell'apprendimento, l'agente Q-Learning completa con successo più del 90% degli episodi, ciò significa che l'agente Q-Learning è in grado, mediante stato ridotto e discretizzato, di completare con successo il task, anche in caso di oggetto in continuo movimento. Le performance ottenute negli ambienti con oggetto in movimento risultano migliori in quanto l'agente apprende un comportamento opportunistico: attende l'oggetto con pinza aperta e lo afferra appena lo raggiunge. Inoltre essendoci un minore rischio di far cadere l'oggetto, se toccato in modo maldestro con la pinza, la probabilità di afferrarlo risulta maggiore. Nell'ambiente robotArm Pipe, il reward al completamento con successo dell'episodio, è scalato rispetto al numero di oscillazioni completate dall'oggetto, con un massimo di reward pari a 500. Si può osservare nel grafico di sinistra che l'agente Q-Learning, pur riuscendo a completare più del 90% degli episodi al termine del training, non è riuscito ad ottenere la stima della funzione ottima, in quanto la media dei total reward ottenuti risulta inferiore a 200, valore definito minimo per determinare la vincita dell'episodio. Ciò significa che l'agente Q-Learning, non è riuscito a stimare correttamente la funzione valore dell'ambiente robotArm Pipe.

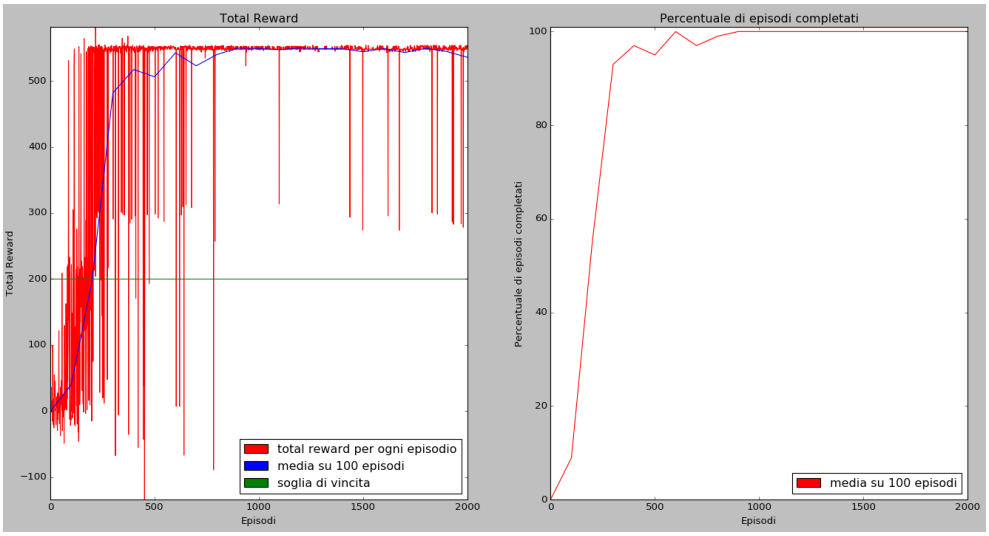

Figura 28 - Grafici delle performance per Deep Q-Learning in robotArm Pipe. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 28, mostrano le performance di Deep Q-Learning in fase di training in ambiente robotArm Pipe. Mediante il grafico di destra si osserva che l'agente Deep Q-Learning, in seguito al training su 500 episodi completa il 100% degli episodi di robotArm Pipe. Osservando il grafico di sinistra, riportante i total reward, si nota che Deep Q-Learning è in grado di completare il task minimizzando il numero di oscillazioni dell'oggetto in quanto il total reward medio ottenuto è superiore a 500, massimo reward terminale ottenibile dall'ambiente robotArm Pipe. Ciò significa che la rete neurale di Deep Q-Learning è riuscita a stimare correttamente la funzione valore dell'ambiente robotArm Pipe, determinando la sequenza di azioni che permette all'agente di ottenere il massimo total reward nell'ambiente robotArm Pipe.

5.2.1 Risultati ottenuti in ambiente robotArm Pipe Hard

Con robotArm Pipe Hard, si identifica una versione più complessa dell'ambiente robotArm Pipe, nella quale ad ogni episodio, la posizione e la direzione di movimento dell'oggetto sull'half pipe, varia in modo casuale. In questo modo, le configurazioni iniziali ad ogni episodio non sono equivalenti tra loro incrementando la variabilità dell'ambiente.

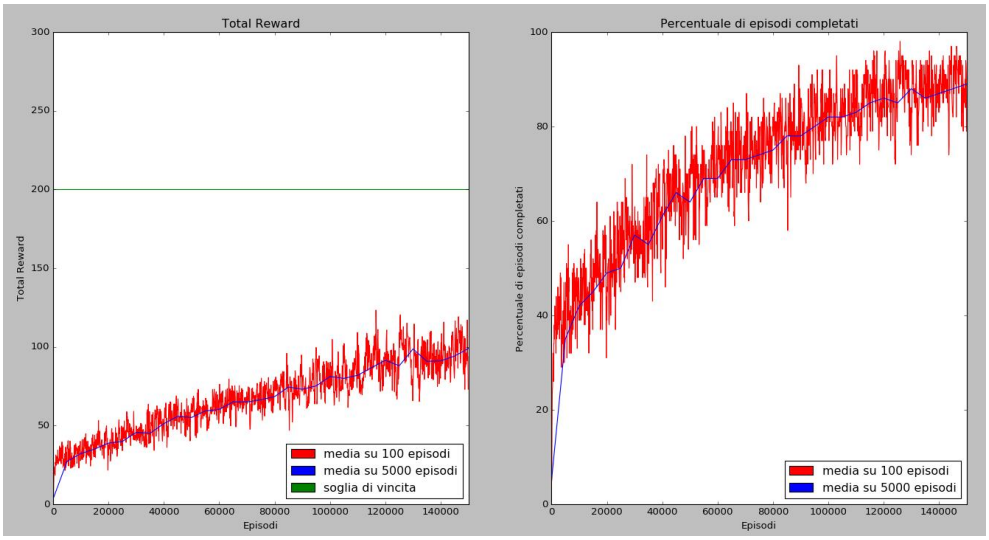

Figura 29 - Grafici delle performance per Q-Learning in robotArm Pipe Hard. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 29, mostrano le performance di Q-Learning in fase di training in ambiente robotArm Pipe Hard. Il grafico di destra mostra come, anche in una situazione più variabile, l'agente Q-Learning è in grado di portare a termine con successo più del 90% degli episodi in seguito al completamento della fase di training. Come in ambiente robotArm Pipe, anche in questo caso l'agente Q-Learning non è in grado di massimizzare il total reward medio, infatti osservando il grafico di sinistra, riportante l'andamento del total reward medio in funzione degli episodi terminati, in prossimità del termine del training, l'agente Q-Learning ottiene reward totali medi non superiori a 100. Analogamente a robotArm Pipe, l'agente Q-Learning riesce ad ottenere un'elevata accuratezza nel completare il task ma non riesce a stimare correttamente la funzione valore dell'ambiente.

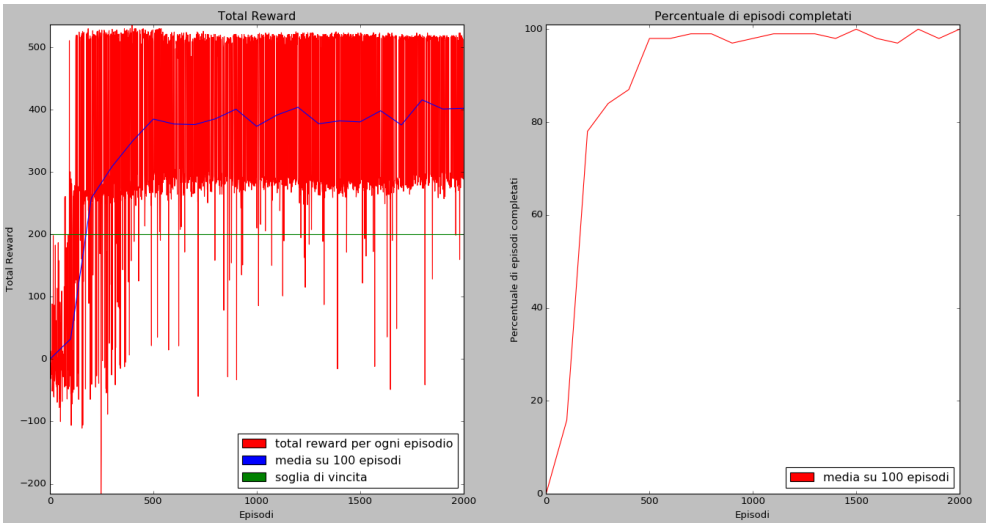

Figura 30 - Grafici delle performance per Deep Q-Learning in robotArm Pipe Hard. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 30, mostrano le performance di Deep Q-Learning in fase di training in ambiente robotArm Pipe Hard. Osservando il grafico di destra, si nota che rispetto a robotArm Pipe, in robotArm Pipe Hard l'agente Deep Q-Learning ha un'accuratezza nel completare correttamente gli episodi leggermente inferiore, ma comunque eccellente, in quanto, dopo il train su 500 episodi riporta un'accuratezza superiore al 98%. Mediante il grafico di sinistra si può osservare

come la variazione casuale della posizione e direzione di movimento iniziale dell'oggetto, incida nel total reward maggiore ottenibile. In media, il total reward ottenuto è 400 ma osservando i total reward per episodio è possibile notare l'oscillazione dei valori tra 500 e 300. L'oscillazione è data dall'iniziale posizione e direzione di movimento casuale dell'oggetto. In ogni caso, l'agente Deep Q-Learning è riuscito a stimare correttamente la funzione valore in quanto completa in media gli episodi minimizzandone il numero di oscillazioni in funzione della posizione e direzione di movimento iniziali dell'oggetto.

5.2.2 Risultati ottenuti in ambiente robotArm Pipe 2 Hard

Con robotArm Pipe 2 Hard, si identifica una versione più complessa dell'ambiente robotArm Pipe Hard, nella quale ad ogni episodio, oltre alla posizione e direzione di movimento dell'oggetto sull'half pipe, anche gli angoli dei tre giunti del braccio meccanico sono inizializzati in posizione casuale. Rappresenta l'ambiente maggiormente stocastico realizzato.

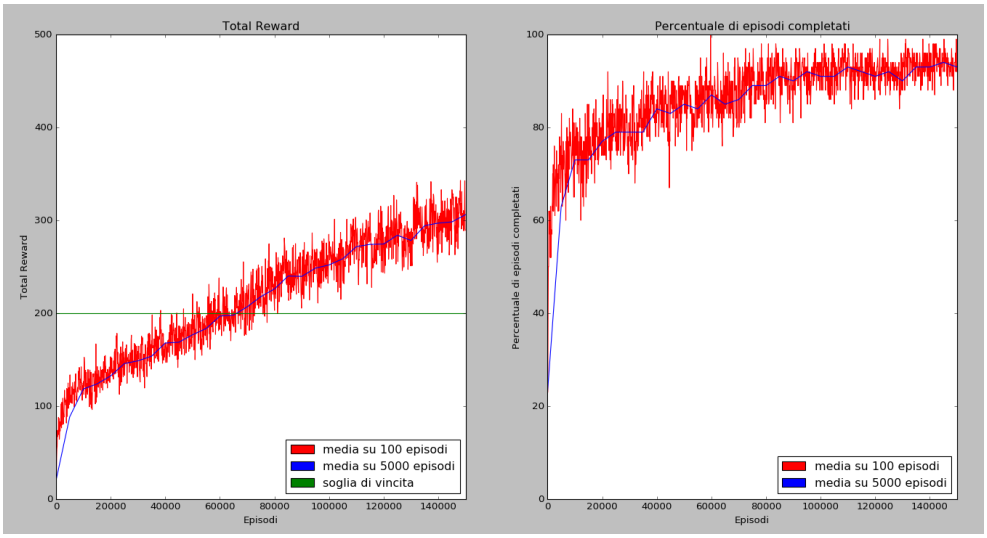

Figura 31 - Grafici delle performance per Q-Learning in robotArm Pipe 2 Hard. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 31, mostrano le performance di Q-Learning in fase di training in ambiente robotArm Pipe 2 Hard. Sorprendentemente, l'agente Q-Learning nell'ambiente più complesso e stocastico realizzato, apprende più

velocemente le dinamiche migliori per afferrare correttamente l'oggetto. Infatti osservando il grafico di destra si nota come in seguito al termine di 10'000 episodi l'agente Q-Learning in ambiente robotArm Pipe 2 Hard ottiene un'accuratezza superiore all'80%. In prossimità del termine della fase di training l'agente Q-Learning ottiene un'accuratezza media superiore al 94%. Osservando il grafico di sinistra riportante il total reward medio ottenuto negli episodi, si nota che a differenza dei risultati ottenuti nelle diverse versioni di robotArm Pipe, in questo caso l'agente, dopo 60'000 episodi, supera la soglia di vincita ottenendo total reward medi superiori a 200. In prossimità del termine della fase di training ottiene total reward medi pari a 320. L'elevata variabilità dell'ambiente, ha consentito all'agente Q-Learning di stimare con più generalità i valori delle coppie stato-azione riuscendo ad ottenere total reward migliori.

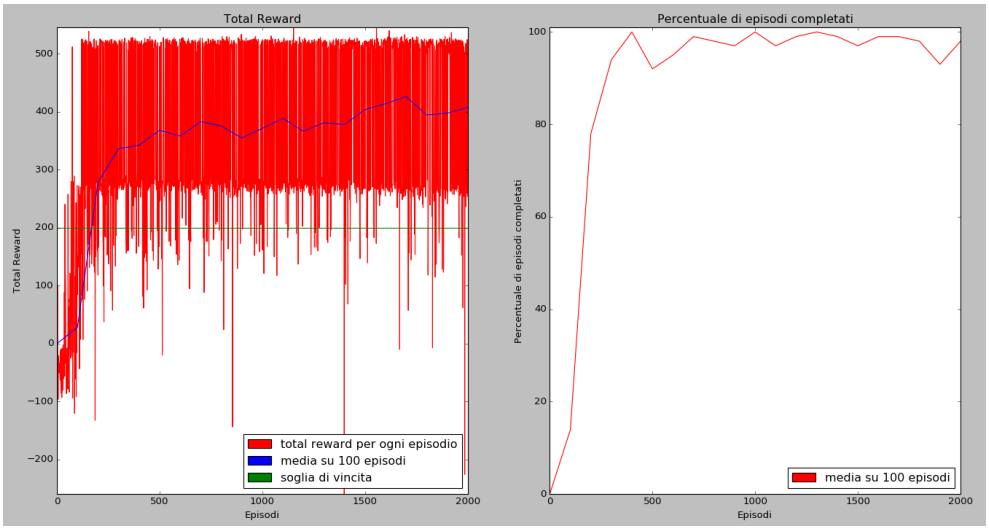

Figura 32 - Grafici delle performance per Deep Q-Learning in robotArm Pipe 2 Hard. A sinistra: andamento del total reward medio in funzione del numero di episodi portati a termine. A destra: andamento della percentuale di episodi completati in funzione del numero di episodi portati a termine.

I grafici riportati in Figura 32, mostrano le performance di Deep Q-Learning in fase di training in ambiente robotArm Pipe 2 Hard. Osservando i grafici si può notare come l'agente Deep Q-Learning eccelle anche nell'ambiente più stocastico realizzato ottenendo un'accuratezza prossima al 100% in seguito a soli 400 episodi portati a termine. Inoltre osservando il grafico di sinistra si nota come Deep Q-Learning riesca ad ottenere un total reward medio pari a 400, equivalente al

massimo valore medio ottenibile nell'ambiente robotArm 2 Hard. Infatti, introducendo variazioni casuali alla posizione iniziale del braccio, i risultati ottenuti risultano simili agli ottenuti in robotArm Hard. Anche per Deep Q-Learning, una maggiore variabilità dello stato iniziale dell'ambiente, ha portato l'algoritmo a determinare più velocemente le dinamiche corrette per afferrare l'oggetto.

5.3 Risultati a confronto

Mettendo a confronto le performance dei due agenti, mediante l'osservazione dei grafici riportati nei paragrafi precedenti relativi alle quattro configurazioni di robotArm implementate, è semplice notare come l'agente Deep Q-Learning riesca ad ottenere risultati migliori rispetto all'agente Q-Learning in ogni configurazione dell'ambiente. Sia dal punto di vista del numero di episodi richiesti per apprendere le dinamiche dell'ambiente, che la media dei total reward ottenuti e l'accuratezza nel portare a termine il task, Deep Q-Learning ottiene risultati migliori rispetto a Q-Learning. In Deep Q-Learning, grazie alla possibilità di utilizzare lo stato completo non discretizzato, ad ogni step l'agente ottiene stati più dettagliati dell'ambiente permettendo un'approssimazione più accurata della funzione valore. Inoltre grazie all'utilizzo della replay memory, ogni aggiornamento del valore dei pesi avviene in funzione dell'errore di stima di 75 diverse coppie stato-azione. L'aggiornamento a mini-batch, utilizzato per l'aggiustamento dei pesi della rete di Deep Q-Learning, porta vantaggi sia dal punto di vista della generalizzazione che dal punto di vista del numero di episodi richiesti per convergere ad una soluzione ottima.

Per avendo performance inferiori, il metodo Q-Learning con tabella dei valori materializzata mediante dizionario, in diverse situazioni riesce ad ottenere buoni risultati nel portare a termine il task raggiungendo valori di accuratezza prossimi al 90%, anche mediante una rappresentazione dello stato degli ambienti ridotta e discretizzata. A sorpresa, Q-Learning ottiene risultati migliori nell'ambiente più complesso e stocastico di robotArm Pipe, rispetto che nelle versioni più semplici. Ciò dimostra che in situazioni complesse, con la giusta generalizzazione ed esplorazione degli stati dell'ambiente, anche in seguito a limitazione e discretizzazione dei parametri di stato, è possibile attraverso il reinforcement learning, far apprendere automaticamente ad un agente autonomo come afferrare un oggetto. In Tabella 8 vengono riepilogate le performance ottenute dagli agenti

nei diversi ambienti. Per ogni test eseguito vengono riportati i valori massimi, in media su 100 episodi consecutivi, di accuratezza e total reward ottenuti dagli agenti. Inoltre, per fornire una misura più generalizzata e veritiera vengono indicati i valori medi ottenuti dagli agenti negli ultimi 500 episodi portati a termine.

Ambiente	Agente	Best Accuracy	Best Total Reward	Media ultimi 500 Accuracy	Media ultimi 500 Total Reward
robotArm Simple	**Q-Learn**	85%	206.41	76.80%	176.97
	DQ-Learn	100%	253.68	96.99%	239.27
robotArm Pipe	**Q-Learn**	98%	106.10	90.80%	99.33
	DQ-Learn	100%	549.77	100%	544.32
robotArm Pipe Hard	**Q-Learn**	98%	98.98	88.20%	94.74
	DQ-Learn	100%	415.82	98.60%	398.60
robotArm Pipe 2Hard	**Q-Learn**	100%	306.15	94.20%	294.84
	DQ-Learn	100%	426.18	97.39%	407.90

Tabella 8 - Tabella di riepilogo dei risultati ottenuti negli ambienti realizzati dai metodi Q-Learning e Deep Q-Learning implementati

Mettendo a confronto il tempo di calcolo medio richiesto per portare a termine un episodio, l'agente Q-Learning richiede in media 56.98ms mentre l'agente Deep Q-Learning richiede 862.5ms. Considerando però la differenza in numero di episodi richiesti in fase di training, l'agente Q-Learning, porta a termine 150000 episodi previsti in media 2 ore e 22 minuti, contro i 28 minuti richiesti in media dall'agente Deep Q-Learning per portare a termine 2000 episodi.

Conclusioni e Sviluppi futuri

Grazie ai risultati sperimentali ottenuti dagli agenti Q-Learning e Deep Q-Learning negli ambienti robotArm, è stato dimostrato come mediante l'apprendimento per rinforzo, un agente autonomo sia in grado di apprendere automaticamente come afferrare un oggetto in posizione statica e dinamica. Sia l'agente più semplice Q-Learning, con materializzazione della tabella mediante dizionario, che l'agente più complesso Deep Q-Learning, il quale sostituisce la tabella con una rete neurale, ottengono ottimi risultati di accuratezza nel completare i tasks, inoltre Deep Q-Learning riesce a stimare correttamente la funzione valore massimizzando il total reward ottenuto al completamento degli episodi; ciò implica che l'agente Deep Q-Learning sia in grado di minimizzazione il numero di step richiesti per raggiungere ed afferrare l'oggetto. I risultati dimostrano come il reinforcement learning rappresenti una tecnologia di apprendimento interessante e adattativa in quanto, mantenendo invariato l'algoritmo, l'agente è in grado di completare diversi task con performance di accuratezza eccellenti anche in presenza di stato dell'ambiente parziale e discretizzato. I recenti risultati pubblicati in letteratura, ottenuti mediante il reinforcement learning [6], più precisamente mediante Q-Learning, hanno destato grande interesse della comunità scientifica in questa area di ricerca, in quanto, le proprietà adattative del metodo potrebbero, in un futuro, rappresentare la base di un'intelligenza artificiale robotica in grado di adattarsi autonomamente nel mondo reale.

Per quanto riguarda le scelte implementative, implementare gli ambienti seguendo le interfacce proposte dal toolkit OpenAI Gym, permette di ampliare la libreria di ambienti proposti dal toolkit in modo da fornire robotArm agli utenti della community come ambiente di test per gli algoritmi di reinforcement learning da loro realizzati. Si noti che robotArm affronta, attraverso un ambiente di simulazione, un problema molto importante in robotica [14]. Dati i risultati incoraggianti ottenuti in ambiente 2D, come sviluppi futuri si potrebbe immaginare una simulazione in versione 3D e successivamente, sviluppare una simile applicazione con un braccio robotico reale. Un ulteriore sviluppo futuro, ispirato a molti lavori recentemente pubblicati in letteratura, consiste nel sostituire la corrente rappresentazione dello stato degli ambienti (vedi Tabella 3) con la matrice

RGB dei pixel ottenuti mediante il rendering dei frame degli ambienti, in modo da determinare se mediante l'utilizzo della scena come rappresentazione dello stato dell'ambiente, l'agente è in grado di apprendere le dinamiche dell'ambiente riuscendo a raggiungere l'obiettivo.

Bibliografia

[1] T. M. Mitchell, Machine Learning, 1997.

[2] D. E. Moriarty, A. C. Schultz e J. J. Grefenstette, «Evolutionary algorithms for reinforcement learning,» *Journal of Artificial Intelligence Research,* 1999.

[3] S. D. R. D. C. W. A. Darrel Whitley, «Genetic Reinforcement Learning for Neurocontrol Problems,» *Springer,* 1993.

[4] Watkins, Ph.D. Thesis, 1989.

[5] P. D. Christopher J.C.H. Watkins, «Technical Note - Q-Learning,» *Machine Learning.*

[6] DeepMind, «Human-level control through deep reinforcement learning,» *Nature,* 2015.

[7] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang e W. Zaremba, «OpenAI Gym,» 2016.

[8] «Python,» [Online]. Available: https://www.python.org/.

[9] E. Catto, «Box2D web site,» [Online]. Available: http://box2d.org/.

[10] pybox2d. [Online]. Available: https://github.com/pybox2d/pybox2d.

[11] «Dict on Python Doc,» [Online]. Available: https://docs.python.org/2/library/stdtypes.html?highlight=dict#dict.

[12] G. Research, «TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,» p. 19, 2015.

[13] N. S. S. Geoffrey Hinton, «Neural Networks for Machine Learning - RMSProp,» [Online]. Available: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[14] N. S. Pollard, «The Grasping Problem: Toward Task-Level Programming for an Articulated Hand,» MIT Artificial Intelligence Laboratory, Massachusetts, 1990.

[15] R. S. Sutton e A. G. Barto, Reinforcement Learning: An Introduction, Cambridge, Massachusetts: The MIT Press, 2012.

Appendice

```
#Implementazione dell'ambiente robotArm Simple

import sys, math,random, numpy as np
import Box2D
import gym

FPS = 35.0
SCALE  = 30.0

VIEWPORT_W = 600
VIEWPORT_H = 400

W = VIEWPORT_W/SCALE
H = VIEWPORT_H/SCALE

DEGTORAD = 0.0174532925199432957
RADTODEG = 57.295779513082320876

GROUND_POLY = [
        (0,H),((1/11.0)*W,H/4),
        ((10/11.0)*W,H/4),(W,H)
]

SHOULDER_POLY = [
        (-14,+60), (-20,50), (-30,0),
        (+30,0), (+20,50), (+14,+60)
        ]

ARM_A_LENGTH = 160
ARM_B_LENGTH = 96
ARM_C_LENGTH = 48

ARM_A_WIDTH = 13
ARM_B_WIDTH = 8
ARM_C_WIDTH = 5

ARM_A_POLY = [
        (-(ARM_A_LENGTH/2-4),+ARM_A_WIDTH),(+(ARM_A_LENGTH/2-4),+ARM_B_WIDTH),
        (+ARM_A_LENGTH/2,+3),(+ARM_A_LENGTH/2,-3),
        (+(ARM_A_LENGTH/2-4),-ARM_B_WIDTH),(-(ARM_A_LENGTH/2-4),-ARM_A_WIDTH),
        (-ARM_A_LENGTH/2,-3),(-ARM_A_LENGTH/2,+3)
        ]

ARM_B_POLY = [
        (-(ARM_B_LENGTH/2-4),+ARM_B_WIDTH),(+(ARM_B_LENGTH/2-4),+ARM_C_WIDTH),
        (+ARM_B_LENGTH/2,+3),(+ARM_B_LENGTH/2,-3),
        (+(ARM_B_LENGTH/2-4),-ARM_C_WIDTH),(-(ARM_B_LENGTH/2-4),-ARM_B_WIDTH),
        (-ARM_B_LENGTH/2,-3),(-ARM_B_LENGTH/2,+3)
        ]



ARM_C_POLY = [
```

```
        (-(ARM_C_LENGTH/2-4),+ARM_C_WIDTH),(+(ARM_C_LENGTH/2-4),+4),
        (+ARM_C_LENGTH/2,+3),(+ARM_C_LENGTH/2,-3),
        (+(ARM_C_LENGTH/2-4),-4),(-(ARM_C_LENGTH/2-4),-ARM_C_WIDTH),
        (-ARM_C_LENGTH/2,-3),(-ARM_C_LENGTH/2,+3)
        ]


ARM_C_POS = [ (x/SCALE,y/SCALE) for x,y in [
        (ARM_C_LENGTH/2-1+2,5),(ARM_C_LENGTH/2+1+2,5),
        (ARM_C_LENGTH/2+1+2,-5),(ARM_C_LENGTH/2-1+2,-5),]
        ]

CLAW_DX_LOW = [
        (0,2),(6,1),
        (6,0),(0,-2)
]

CLAW_DX_HIGH = [
        (6,1),(8,-3),(6,0)
]

BALL_RADIUS = 6

#Categories for Collision filtering
ARM_CAT = 0x0010
POS_CAT = 0x8000
CLAW_CAT = 0x0004
PEDESTAL_CAT = 0x0008
BALL_CAT = 0x0020
ALL_CAT = 0xFFFF
GROUND_CAT = 0x0001
NOTHING_CAT = 0x0000

#Motors speed
MOTORSPEED=0.8
CLAWMOTORSPEED=8

class ContactDetector(contactListener):
        def __init__(self, env):
                contactListener.__init__(self)
                self.env = env
        def BeginContact(self, contact):
                if self.env.clawDx in [contact.fixtureA.body,contact.fixtureB.body] and
                   self.env.ball in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.clawDxCnt+=1
                elif self.env.clawSx in [contact.fixtureA.body,contact.fixtureB.body] and
                     self.env.ball in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.clawSxCnt+=1
                elif self.env.ball in [contact.fixtureA.body,contact.fixtureB.body] and
                     self.env.ground in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.ground_contact = True

        def EndContact(self, contact):
                if self.env.clawDx in [contact.fixtureA.body,contact.fixtureB.body] and
                   self.env.ball in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.clawDxCnt-=1
                elif self.env.clawSx in [contact.fixtureA.body,contact.fixtureB.body] and
                     self.env.ball in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.clawSxCnt-=1
                elif self.env.ball in [contact.fixtureA.body,contact.fixtureB.body] and
                     self.env.ground in [contact.fixtureA.body,contact.fixtureB.body]:
                        self.env.ground_contact = False
```

```
class RoboticArmSimple(gym.Env):
	metadata = {
		'render.modes': ['human', 'rgb_array'],
		'video.frames_per_second' : FPS
	}

	def __init__(self):
		self._seed()
		self.viewer = None

		self.world = Box2D.b2World()
		self.ground = None

		high = np.array([np.inf]*6)
		self.observation_space = spaces.Box(-high, high)
		self.action_space = spaces.Discrete(8)

		self.currAngle = 0
		self._reset()


	def _getDistance(self):
		shapeRAP = [self.roboticArmPosition.body.transform*v for v in self.roboticArmPosition.shape]
		centroidRAP = [sum(c.x for c in shapeRAP)/len(shapeRAP),sum(c.y for c in shapeRAP)/len(shapeRAP)]
		distance = np.absolute(self.ball.position.x-centroidRAP[0])+np.absolute(self.ball.position.y-centroidRAP[1])
		return distance

	def _getDeltaXY(self):
		shapeRAP = [self.roboticArmPosition.body.transform*v for v in self.roboticArmPosition.shape]
		centroidRAP = [sum(c.x for c in shapeRAP)/len(shapeRAP),sum(c.y for c in shapeRAP)/len(shapeRAP)]
		return self.ball.position.x-centroidRAP[0], self.ball.position.y-centroidRAP[1]

	def _seed(self, seed=None):
		self.np_random, seed = seeding.np_random(seed)
		return [seed]

	def _destroy(self):
		if not self.ground: return
		self.world.contactListener = None
		self.world.DestroyBody(self.ground)
		self.ground = None
		self.world.DestroyBody(self.armA)
		self.armA= None
		self.world.DestroyBody(self.armB)
		self.armB= None
		self.world.DestroyBody(self.armC)
		self.armC= None
		self.world.DestroyBody(self.shoulder)
		self.shoulder= None
		self.world.DestroyBody(self.clawDx)
		self.clawDx= None
		self.world.DestroyBody(self.clawSx)
		self.clawSx= None
		self.world.DestroyBody(self.ball)
		self.ball = None
		self.reward = None
		self.done = None
		self.world.DestroyBody(self.pedestal)
		self.pedestal=None
		self.roboticArmPosition=None
		self.timeClosed = None
		self.prevDistance = None
```

```
def _reset(self):
        self._destroy()
        self.world.contactListener_keepref = ContactDetector(self)
        self.world.contactListener = self.world.contactListener_keepref
        self.ground_contact = False
        self.reward = 0
        self.done = False
        self.clawSxCnt = 0
        self.clawDxCnt = 0
        self.captured = False
        self.clawOpened = True
        self.steps = 0
        self.timeClosed = 0
        self.prevDistance = None

        self.InitialBallX = random.randint(260,350)
        self.InitialBallY = random.randint(1,150)

        #terrain def
        self.ground = self.world.CreateStaticBody()
        self.sky_polys = []
        for i in range(len(GROUND_POLY)-1):
                p1 = GROUND_POLY[i]
                p2 = GROUND_POLY[i+1]
                self.ground.CreateEdgeFixture(
                        vertices=[p1,p2],
                        density=0,
                        friction=0.1)

        #shoulder def
        self.shoulder = self.world.CreateStaticBody(
                position=(W/4,H/4),
                angle=0.0,
                fixtures=fixtureDef(
                        shape=polygonShape(vertices=[(x/SCALE,y/SCALE) for x,y in SHOULDER_POLY ]),
                        density=0.2,
                        friction=0.1,
                        userData = True))
        self.shoulder.color1 = (0,1,0)
        self.shoulder.color2 = (0,0.5,0)
        deltaAngleArmA = random.randint(0,60)

        #part A of arm def
        self.armA = self.world.CreateDynamicBody(
                position = (W/4,H/3),
                angle=(90)*DEGTORAD,
                fixtures = fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE,y/SCALE) for x,y in ARM_A_POLY ]),
                        density=0.2,
                        friction=0.1,
                        categoryBits=ARM_CAT,
                        maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True)
                        )
        self.armA.color1 = (1,1,1)
        self.armA.color2 = (0,0,0)

        self.armA.CreateFixture(
                fixtureDef(
                        shape=circleShape(radius=ARM_A_WIDTH/SCALE, pos=((-ARM_A_LENGTH+7)/2/SCALE,0)),
                        density=0.0,
```

```
                        friction=0.0,
                        categoryBits=ARM_CAT,
                        maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True)
)

#joint shoulder-partA def
shoulderJoint = revoluteJointDef(
        bodyA = self.shoulder,
        bodyB = self.armA,
        localAnchorA=(0/SCALE,50/SCALE),
        localAnchorB=((-ARM_A_LENGTH+7)/2/SCALE,0/SCALE),
        enableMotor=True,
        enableLimit=True,
        maxMotorTorque=400.0,
        motorSpeed=0
        )
shoulderJoint.lowerAngle = 30*DEGTORAD-self.armA.angle
shoulderJoint.upperAngle = 90*DEGTORAD-self.armA.angle
self.shoulder.joint = self.world.CreateJoint(shoulderJoint)

#part B of arm def
self.armB = self.world.CreateDynamicBody(
        position = (W/4,H/3),
        angle=0,
        fixtures = fixtureDef(
                shape=polygonShape(vertices=[ (x/SCALE,y/SCALE) for x,y in ARM_B_POLY ]),
                density=0.2,
                friction=0.1,
                categoryBits=ARM_CAT,
                maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                restitution=0.0,
                userData = True)
                )
self.armB.color1 = (1,1,1)
self.armB.color2 = (0,0,0)

self.armB.CreateFixture(
        fixtureDef(
                shape=circleShape(radius=ARM_B_WIDTH/SCALE, pos=((-ARM_B_LENGTH+7)/2/SCALE,0)),
                density=0.0,
                friction=0.0,
                categoryBits=ARM_CAT,
                maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                restitution=0.0,
                userData = True)
)

#joint armA - armB def
elbowJoint = revoluteJointDef(
        bodyA = self.armA,
        bodyB = self.armB,
        localAnchorA=((ARM_A_LENGTH/2-4)/SCALE,0/SCALE),
        localAnchorB=((-ARM_B_LENGTH+7)/2/SCALE,0/SCALE),
        enableMotor=True,
        enableLimit=True,
        maxMotorTorque=400.0,
        motorSpeed=0
        )
elbowJoint.lowerAngle = -120*DEGTORAD+self.armA.angle
elbowJoint.upperAngle = 0*DEGTORAD+self.armA.angle
self.elbowjoint = self.world.CreateJoint(elbowJoint)
```

```
#part C of arm def
self.armC = self.world.CreateDynamicBody(
            position = (W/4,H/2),
            angle=0,
            fixtures = fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE,y/SCALE) for x,y in ARM_C_POLY ]),
                        density=0.2,
                        friction=0.1,
                        categoryBits=ARM_CAT,
                        maskBits=0x001|BALL_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True)
                        )

self.armC.CreateFixture(
            fixtureDef(
                        shape=circleShape(radius=ARM_C_WIDTH/SCALE, pos=(-(ARM_C_LENGTH/2-4)/SCALE,0)),
                        density=0.0,
                        friction=0.0,
                        categoryBits=ARM_CAT,
                        maskBits=GROUND_CAT|BALL_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True)
)

#fixture on armC used to calculate DeltaX and DeltaY every step
self.roboticArmPosition=self.armC.CreateFixture(
            fixtureDef(
                        shape=polygonShape(vertices=ARM_C_POS),
                        density=0.0,
                        friction=0.0,
                        categoryBits=POS_CAT,
                        maskBits=NOTHING_CAT,  # collide with nothing
                        restitution=0.0,
                        userData = False)
            )

self.armC.color1 = (1,1,1)
self.armC.color2 = (0,0,0)

#joint armB-armC def
wristJoint = revoluteJointDef(
            bodyA = self.armB,
            bodyB = self.armC,
            localAnchorA=((ARM_B_LENGTH/2-4)/SCALE,0/SCALE),
            localAnchorB=(-(ARM_C_LENGTH/2-4)/SCALE,0/SCALE),
            enableMotor=True,
            enableLimit=True,
            maxMotorTorque=400.0,
            motorSpeed=0
            )
wristJoint.lowerAngle = -90*DEGTORAD+self.armB.angle
wristJoint.upperAngle = 0*DEGTORAD+self.armB.angle
self.wristjoint = self.world.CreateJoint(wristJoint)


#clawDx def
self.clawDx = self.world.CreateDynamicBody(
            position = (W/4,H/2),
            angle=90*DEGTORAD,
            fixtures = fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE*2.5,y/SCALE*2.5) for x,y in CLAW_DX_HIGH ]),
```

```
                        density=0.2,
                        friction=0.0,
                        categoryBits=CLAW_CAT,
                        maskBits=BALL_CAT|GROUND_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True),
                bullet = True
                        )
self.clawDx.CreateFixture(
                fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE*2.5,y/SCALE*2.5) for x,y in CLAW_DX_LOW ]),
                        density=0.2,
                        friction=0.0,
                        categoryBits=CLAW_CAT,
                        maskBits=BALL_CAT|GROUND_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True))

self.clawDx.color1 = (1,0,0)
self.clawDx.color2 = (1,0,0)

#joint armC-clawDx def
clawDxJoint = revoluteJointDef(
                bodyA = self.armC,
                bodyB = self.clawDx,
                localAnchorA=(18/SCALE,0/SCALE),
                localAnchorB=(0/SCALE,0/SCALE),
                enableMotor=True,
                enableLimit=True,
                maxMotorTorque=300.0,
                motorSpeed=0,
                )
clawDxJoint.lowerAngle = -58*DEGTORAD
clawDxJoint.upperAngle = 0*DEGTORAD
self.clawDxJoint = self.world.CreateJoint(clawDxJoint)

#clawSx def
self.clawSx = self.world.CreateDynamicBody(
                position = (W/4,H/2),
                angle=-90*DEGTORAD,
                fixtures = fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE*2.5,-y/SCALE*2.5) for x,y in CLAW_DX_HIGH ]),
                        density=0.2,
                        friction=0.0,
                        categoryBits=CLAW_CAT,
                        maskBits=BALL_CAT|GROUND_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True),
                bullet = True)

self.clawSx.CreateFixture(
                fixtureDef(
                        shape=polygonShape(vertices=[ (x/SCALE*2.5,-y/SCALE*2.5) for x,y in CLAW_DX_LOW ]),
                        density=0.2,
                        friction=0.0,
                        categoryBits=CLAW_CAT,
                        maskBits=BALL_CAT|GROUND_CAT|PEDESTAL_CAT,
                        restitution=0.0,
                        userData = True))

self.clawSx.color1 = (1,0,0)
self.clawSx.color2 = (1,0,0)
```

```
        clawSxJoint = revoluteJointDef(
                bodyA = self.armC,
                bodyB = self.clawSx,
                localAnchorA=(18/SCALE,0/SCALE),
                localAnchorB=(0/SCALE,0/SCALE),
                enableMotor=True,
                enableLimit=True,
                maxMotorTorque=300.0,
                motorSpeed=0,
                )
        clawSxJoint.lowerAngle = 0*DEGTORAD
        clawSxJoint.upperAngle = 58*DEGTORAD
        self.clawSxJoint = self.world.CreateJoint(clawSxJoint)

        #Pedestal def
        self.pedestal = self.world.CreateStaticBody(
                position=(self.InitialBallX/SCALE,H/4),
                angle=0.0,
                fixtures=fixtureDef(
                        shape=polygonShape(vertices=[(x/SCALE,y/SCALE) for x,y in
                                [(-10,0),(10,0),(10,self.InitialBallY),(-10,self.InitialBallY)] ]),
                        density=2.0,
                        friction=0.1,
                        categoryBits=PEDESTAL_CAT,
                        maskBits = ALL_CAT,
                        userData = True))
        self.pedestal.color1 = (1,1,0)
        self.pedestal.color2 = (0.8,0.8,0)

        #ball to grab with claw def
        self.ball = self.world.CreateDynamicBody(
                position = (self.InitialBallX/SCALE,(self.InitialBallY+BALL_RADIUS*2)/SCALE+H/4),
                angle=0.0,
                fixtures = fixtureDef(
                        shape=circleShape(radius=BALL_RADIUS/SCALE, pos=(0,0)),
                        density=0.1,
                        friction=0.2,
                        categoryBits=BALL_CAT,
                        maskBits=CLAW_CAT | GROUND_CAT | ARM_CAT | BALL_CAT | PEDESTAL_CAT,
                        restitution=0,
                        userData = True)
                        )
        self.ball.color1 = (0,0,1)
        self.ball.color2 = (0,1,1)


        self.drawlist = [self.shoulder] + [self.armA] + [self.armB] +[self.armC] + [self.clawSx] + [self.clawDx] + [self.pedestal] +[self.ball]
        return self._step(np.array([0,0]) if False else 0)[0]


def _step(self, action):

        self.steps+=1
        self.shoulder.joint.motorSpeed= 0
        self.wristjoint.motorSpeed= 0
        self.elbowjoint.motorSpeed= 0

        assert self.action_space.contains(action), "%r (%s) invalid " % (action,type(action))

        if(action != 0):
                if(action == 1):
                        self.shoulder.joint.motorSpeed= +MOTORSPEED
                if(action == 2):
```

```
                self.shoulder.joint.motorSpeed= -MOTORSPEED
        if(action == 3):
                self.elbowjoint.motorSpeed= +MOTORSPEED
        if(action == 4):
                self.elbowjoint.motorSpeed= -MOTORSPEED
        if(action == 5):
                self.wristjoint.motorSpeed=+MOTORSPEED
        if(action == 6):
                self.wristjoint.motorSpeed=-MOTORSPEED
        if(action == 7):
                if self.clawOpened:
                        self.timeClosed+=1
                self.clawOpened = not self.clawOpened

if(self.clawOpened):
        if self.clawDxJoint.angle>=self.clawDxJoint.upperLimit and self.clawSxJoint.angle <= self.clawSxJoint.lowerLimit:
                self.clawDxJoint.motorSpeed=0
                self.clawSxJoint.motorSpeed=0
        else:
                self.clawDxJoint.motorSpeed=CLAWMOTORSPEED
                self.clawSxJoint.motorSpeed=-CLAWMOTORSPEED
else:
        if self.clawDxJoint.angle<=self.clawDxJoint.lowerLimit and self.clawSxJoint.angle >= self.clawSxJoint.upperLimit:
                self.clawDxJoint.motorSpeed=0
                self.clawSxJoint.motorSpeed=0
        else:
                self.clawDxJoint.motorSpeed=-CLAWMOTORSPEED
                self.clawSxJoint.motorSpeed=+CLAWMOTORSPEED

self.world.Step(1.0/FPS, 6*30, 2*30)

#ball captured checks
if self.clawDxCnt+self.clawSxCnt >= 3 and not self.captured:
        self.captured = not self.captured
elif self.clawDxCnt+self.clawSxCnt == 0 and self.captured:
        self.captured = not self.captured

#Reward
self.reward = 0
if not self.done:
        if self.captured:
                self.reward = 200
                self.done = True
        elif self.ground_contact:
                self.reward = 0
                self.done = True
        else:
                curDistance = round(self._getDistance(),2)
                if curDistance < self.prevDistance:
                        self.reward = 1+(not self.clawOpened)*(-.2)
                else:
                        self.reward = -1+(not self.clawOpened)*(-.2)
                if self.steps >= 500: self.done = True

self.prevDistance = round(self._getDistance(),2)

deltaX,deltaY = self._getDeltaXY()
state = [
        self.clawOpened,
        self.shoulder.joint.angle,
        self.elbowjoint.angle,
        self.wristjoint.angle,
        deltaX,
```

```
                deltaY
                ]

        assert len(state)==6
        return np.array(state), self.reward, self.done, {}

def _render(self, mode='human', close=False):
        if close:
                if self.viewer is not None:
                        self.viewer.close()
                        self.viewer = None
                return

        if self.viewer is None:
                self.viewer = rendering.Viewer(VIEWPORT_W, VIEWPORT_H)
                self.viewer.set_bounds(0, VIEWPORT_W/SCALE, 0, VIEWPORT_H/SCALE)

        self.viewer.draw_polygon(GROUND_POLY, color=(0,0,0))

        for obj in self.drawlist:
                for f in obj.fixtures:
                        trans = f.body.transform
                        if type(f.shape) is circleShape:
                                t = rendering.Transform(translation=trans*f.shape.pos)
                                self.viewer.draw_circle(f.shape.radius, 20, color=obj.color1).add_attr(t)

                                if obj.color1==(1,1,1):
                                        self.viewer.draw_circle((f.shape.radius*0.6), 20, color=(0,0,0)).add_attr(t)

                                self.viewer.draw_circle(f.shape.radius, 20, color=obj.color2, filled=False, linewidth=2).add_attr(t)
                        else:
                                if(f.userData==True):
                                        path = [trans*v for v in f.shape.vertices]
                                        self.viewer.draw_polygon(path, color=obj.color1)
                                        path.append(path[0])
                                        self.viewer.draw_polyline(path, color=obj.color2, linewidth=2)

        return self.viewer.render(return_rgb_array = mode=='rgb_array')
```