ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

DEEP REINFORCEMENT LEARNING CON PYTORCH

Elaborato in
MACHINE LEARNING

Relatore
Prof. ALESSANDRO RICCI

Presentata da
FRANCESCO PIANO

Anno Accademico 2020 – 2021

Alla mia famiglia per avermi sempre sostenuto durante questo percorso.

Indice

Introduzione

Il Reinforcement Learning è un campo di ricerca del Machine Learning in cui la risoluzione di problemi da parte di un agente avviene scegliendo l'azione più idonea da eseguire attraverso un processo di apprendimento iterativo, in un ambiente dinamico che lo incentiva tramite ricompense. Il Deep Learning, anch'esso approccio del Machine Learning, sfruttando una rete neurale artificiale è in grado di applicare metodi di apprendimento per rappresentazione allo scopo di ottenere una struttura dei dati più idonea ad essere elaborata. Solo recentemente il Deep Reinforcement Learning, creato dalla combinazione di questi due paradigmi di apprendimento, ha permesso di risolvere problemi considerati prima intrattabili riscuotendo un notevole successo e rinnovando l'interesse dei ricercatori riguardo l'applicazione degli algoritmi di Reinforcement Learning. Con questa tesi si è voluto approfondire lo studio del Reinforcement Learning applicato a problemi semplici, per poi esaminare come esso possa superare i propri limiti caratteristici attraverso l'utilizzo delle reti neurali artificiali, in modo da essere applicato in un contesto di Deep Learning. Si introdurrà quindi un ambiente di learning, OpenAI gym, nel quale affrontare il problema del Cart Pole dapprima nel Reinforcement learning, poi nel Deep Reinforcement Learning: l'agente dovrà mantenere bilanciato un palo fissato a un carrello mobile attraverso un giunto impedendone la caduta tramite spostamenti dello stesso carrello in avanti o indietro. Allo scopo di risolvere questo problema verranno quindi illustrati i due algoritmi: Q-learning che si avvale di una tabella a più dimensioni per memorizzare le coppie stato-azione e rispettivi q-value e Deep Q-learning che, in sostituzione della tabella, sfrutta una rete neurale artificiale come approssimatore di funzione per la funzione valore o la policy. Per sviluppare l'algoritmo Q-learning si utilizzerà Numpy, una libreria per il calcolo scientifico che consente al linguaggio Python di ottenere una capacità computazionale al pari di linguaggi come il C e il Fortran mantenendo la sua intrinseca facilità di utilizzo e apprendimento. Per utilizzare le reti neurali invece si ricorrerà a PyTorch, una libreria attualmente molto usata per il calcolo scientifico e il Machine Learning, sviluppata da Facebook. La struttura dati fondamentale di PyTorch è il Tensore, simile ad un ndarrays (array multidimensionale) di Numpy ma in grado di offrire prestazioni migliori

per quanto riguarda l'accesso e le operazioni sugli elementi degli array. Prima di trattare gli algoritmi sviluppati e le tecnologie messe in campo, la tesi introduce i concetti di base di Machine Learning, Reinforcement Learning e Deep Learning, infine illustra i risultati ottenuti dagli algoritmi Q-learning e Deep Q-learning nella risoluzione del problema Cart Pole nella sua versione base.

Capitolo 1

Introduzione al Reinforcement Learning

Il Reinforcement learning è una tecnica di apprendimento autonomo in grado di determinare le azioni da eseguire in risposta a specifici eventi al fine di massimizzare una ricompensa attraverso un feedback numerico. In base a questo un agente non saprà a priori quale azione eseguire ma sarà incentivato ad apprendere, attraverso tentativi, la più idonea in modo da incentivare questo segnale.
Come sottolinea la composizione del termine stesso, il Deep Reinforcement Learning (DRL) oggetto della presente trattazione, è costituito dalla combinazione di teorie inerenti il Reinforcement Learning e il Deep Learning. Questo campo di ricerca del Machine Learning viene impiegato per risolvere una vasta gamma di task che devono assumere decisioni complesse impiegando un'intelligenza simile a quella umana, finora fuori dalla portata di una macchina. Al fine di analizzarne la complessità, in un'ottica di continua evoluzione della materia, si rende quindi necessario introdurre sommariamente le basi delle teorie dell'apprendimento che ne costituiscono la struttura fondativa dettate dal Machine Learning, dal Reinforcement Learning e dal Deep Learning.

1.1 Machine Learning

Il Machine Learning vede i suoi albori nell'Inghilterra Vittoriana per ricevere una svolta decisiva grazie alle innovative teorie di Alan Turing il quale, nel 1950, giunse ad ipotizzare la necessità di realizzare algoritmi specifici per realizzare macchine in grado di apprendere: nell'articolo *"Computing Machinery and Intelligence"* introdusse i concetti base per modellare un'intelligenza artificiale.
Quando si parla di Machine Learning si affronta quindi una branca dell'infor-

matica strettamente correlata all'AI in cui il focus si sposta dal programmatore alla macchina stessa: non è più l'essere umano che tenta di codificare algoritmi al fine di trovare soluzioni ad un determinato problema considerando dati iniziali e regole prefissate ma, al contrario, è la macchina che, in base ai dati di input e alle risposte fornitele, determina le regole per risolvere il problema tramite un processo di apprendimento automatico. Come meglio esplicitato da Arthur Samuel nel 1959 [11] il Machine Learning può essere definito *"field of study that gives computers the ability to learn without being explicitly programmed"*.
Teoria della probabilità e statistica caratterizzano la basi del Machine Learning che, ad oggi, trova applicazione in diversi ambiti della società moderna, arrivando ad influenzare le scelte dei suoi utilizzatori grazie allo sviluppo di intelligenza artificiale in grado di intercettare e risolvere problematiche quali, ad esempio, il riconoscimento di un oggetto all'interno di un'immagine, la trascrizione di messaggi audio in formato testuale, il posizionamento degli annunci pubblicitari personalizzati per ogni utente:
"Machine Learning is about learning from data and making predictions and/or decisions." [6]
Un algoritmo di machine learning si compone di dataset, modello, funzione d'errore e procedura di ottimizzazione: il dataset viene diviso in diverse parti non sovrapponibili, una per il training dell'algoritmo e una per la sua validazione; la funzione d'errore misura le performance del modello rispetto al livello di accuratezza dei risultati prodotti dall'algoritmo.
Le modalità di apprendimento e di determinazione delle regole ha portato oggi all'individuazione di tre categorie che differiscono non solo per gli algoritmi utilizzati, ma soprattutto per lo scopo per cui sono realizzate le macchine stesse: apprendimento supervisionato, non supervisionato e Reinforcement Learning.

L'apprendimento supervisionato consiste nel fornire all'algoritmo una serie di dati di input e di informazioni relative ai risultati desiderati impostando come obiettivo quello che il sistema identifichi una regola generale che associ l'input all'output corretto. La classificazione e la regressione sono due tipi di apprendimento supervisionato che fornisco rispettivamente come output una categorizzazione e una quantità numerica.

Nell'apprendimento non supervisionato vengono invece forniti al sistema solo set di dati, senza alcuna indicazione del risultato considerato: l'obiettivo in questo caso è quello di risalire a schemi e modelli nascosti ovvero identificare negli input una logica di struttura senza che questi siano preventivamente etichettati. Un classico problema di apprendimento non supervisionato è l'apprendimento per rappresentazione il cui scopo è trovare una rappresentazione

che preservi quante più informazioni possibili del dato originale semplificandone però l'accesso al contenuto informativo.

L'apprendimento per rinforzo, a differenza dei precedenti, opera tramite feedback di valutazione ma senza meccanismi di supervisione. In questo metodo il sistema interagisce con un ambiente dinamico (per ottenere i dati in input) al fine di raggiungere un obiettivo per il quale riceve una "ricompensa", imparando anche dagli errori. Il comportamento e le prestazioni del sistema sono determinate da una serie di apprendimenti basati quindi su ricompense e rilevazioni di errori.

1.2 Reinforcement Learning

Come accennato, il Reinforcement Learning o apprendimento per rinforzo (RL) è uno dei tre paradigmi base del Machine Learning. Con Reinforcement Learning (RL) si definisce quindi una tecnica di apprendimento autonomo atta a creare degli agenti, che operano in un certo environment, in grado di risolvere task basandosi su dati che variano nel tempo: ciò significa che tali agenti avranno un comportamento dinamico, in relazione ai dati elaborati, per cui le azioni da essi eseguite in un dato istante saranno influenzate da quelle effettuate in precedenza.
Traendo spunto dalle teorie comportamentiste sull'apprendimento per prove ed errori di Edward Thorndike e su condizionamenti operanti e rinforzi positivi introdotti da Burrhus F. Skinner all'inizio del XXI secolo, la risoluzione di un determinato task si ottiene addestrando l'algoritmo, che rappresenta l'agente, attraverso ricompense al raggiungimento di obiettivi prefissati e disincentivando invece comportamenti non desiderabili. Contrariamente a quanto il linguaggio naturale potrebbe suggerire, col termine ricompensa si intende sia una quantità positiva che negativa: l'agente avrà quindi come scopo primario quello di massimizzarla apprendendo delle abilità elementari che lo stimolino in tal senso.
Rispetto al paradigma dell'apprendimento supervisionato il reinforcement learning differisce quindi in quanto non necessita di un set di dati di confronto per poter sviluppare conoscenza, ponendo l'accento sul corretto bilanciamento dell'esplorazione di scelte non ancora effettuate e la conoscenza di quelle già affrontate.
Le tecniche di apprendimento per rinforzo si applicano quindi alla risoluzione di task di controllo. Una delle prime strategie di reinforcement learning fu introdotta nel 1957 da Richard Bellman e denominata dynamic programming

(DP): in tale casistica il problema viene scomposto in sotto problemi di dimensione sempre minore e di più facile risoluzione che conducono in ultima istanza alla risoluzione del problema principale. Questa metodologia risultò però non sempre applicabile in maniera generalizzata a tutti i task di controllo in quanto l'insorgere di eventi inattesi ne andava a limitare la funzionalità, inoltre si riscontrò un ulteriore vincolo nel livello di conoscenza posseduto dell'environment che rendeva necessario variare la strategia di risoluzione del problema. Per tali casistiche si rese necessario ricorrere ad una strategia "per tentativi" (trial and error) che generalmente si riconduce ai metodi Monte Carlo, includenti una vasta classe di algoritmi computazionali che effettuano un campionamento casuale dell'environment per ottenere risultati numerici. L'esperienza ha dimostrato che la strategia più idonea nell'affrontare task di reinforcement learning è quella che combina l'applicazione di metodi per tentativi, da cui possiamo ricavare conoscenza dell'environment, con quella di scomposizione in sotto problemi noti. Per comprendere quali problemi il Reinforcement Learning è in grado di risolvere possiamo considerare alcuni esempi ed applicazioni che ne hanno favorito lo sviluppo:

- L'esecuzione di una mossa da parte di un giocatore di scacchi: la scelta è dettata sia dal risultato di una strategia volta ad anticipare mosse e contromosse dell'avversario sia dall'intuizione momentanea volta al desiderio di una particolare posizione.
- L'ottimizzazione dei parametri in tempo reale di una raffineria di petrolio da parte di un controllore: tale operazione consiste nel perfezionare il trade-off rendimento/costi/qualità basandosi su specifici costi marginali, senza necessariamente rispettare i regolamenti originariamente forniti dagli ingegneri.
- L'apprendimento delle abilità motorie da parte di una gazzella: qualche minuto dopo essere nata si alza in piedi, dopo qualche ora è già in grado di correre a 20 miglia all'ora;
- La decisione di un robot aspirapolvere di iniziare a pulire una nuova stanza oppure rientrare alla base per ricaricarsi: fattore discriminante sarà da un lato l'attuale livello di carica delle batterie e dall'altro la velocità e facilità con cui in passato è stato in grado di raggiungere la stazione di ricarica.

Tutti questi esempi condividono delle caratteristiche basilari facili da individuare: tra le altre, tutti presentano un'interazione tra un agente attivo, capace di prendere decisioni, e l'ambiente in cui è immerso, nel quale cerca di raggiungere un obiettivo in contrasto con l'incertezza dello stesso. Le azioni che l'agente esegue permettono di simulare lo stato futuro dell'ambien-

te. Riprendendo gli esempi riportati questo si traduce nella previsione della successiva posizione di un pezzo sulla scacchiera, del livello delle riserve della raffineria, della posizione successiva del robot e del futuro livello di carica: in tal modo l'agente potrà capire quali azioni e opportunità saranno disponibili in seguito. La scelta corretta richiede però anche una considerazione indiretta, tale da tener conto delle conseguenze a posteriori delle azioni e di indurre capacità di pianificazione e previsione. Allo stesso tempo, l'effetto delle azioni non può essere totalmente determinato e per tale ragione l'agente deve monitorare l'ambiente e reagire in modo appropriato. Tutti gli esempi elencati comprendono obiettivi espliciti con cui l'agente può giudicare i suoi progressi una volta che questi vengono raggiunti: il giocatore di scacchi riconosce quando ha vinto o meno, il controllore della raffineria sa quanto petrolio viene prodotto, la gazzella percepisce un'imminente caduta ed infine il robot stima quando la batteria sarà scarica. L'agente può però anche usare la propria esperienza per migliorare le sue performance nel tempo in modo che, ad esempio, durante il gioco si raffini l'intuizione riguardo la valutazione delle successive posizioni degli scacchi oppure, come nel caso della gazzella, possa migliorare l'efficienza della corsa. La conoscenza che l'agente porta all'inizio della risoluzione del task, che può essere accumulata da task simili svolti precedentemente oppure costruita da specifica all'interno dell'agente stesso, influenza ciò che è utile o facile da apprendere mentre l'interazione con l'ambiente è essenziale per correggere il suo comportamento al fine di adattarsi alle specifiche casistiche del task in esecuzione.

1.3 Il pole balancing

Un tipico problema volto ad applicare le tecniche di Reinforcement Learning è il pole balacing: il sistema è illustrato in figura 1.1 ed è composto da un carrello (cart) e da una barra (pole) verticale ad esso connessa attraverso un giunto passivo avente soli due gradi di libertà. L'obiettivo sarà quello di mantenere l'angolo di inclinazione θ entro certi limiti impedendo che la barra cada: questo si realizza muovendo il carrello nelle due direzioni consentite, ossia a destra o sinistra. Questo task può essere trattato in maniera episodica, in ogni episodio il pole viene posto in verticale e si tenta di mantenerne il bilanciamento ripetendo questa procedura su un certo numero di episodi. Il reward in questo caso sarà $+1$ per ogni istante di tempo in cui il pole non cade e quindi per ciascun episodio sarà rappresentato dal numero totale di step di bilanciamento effettuati.

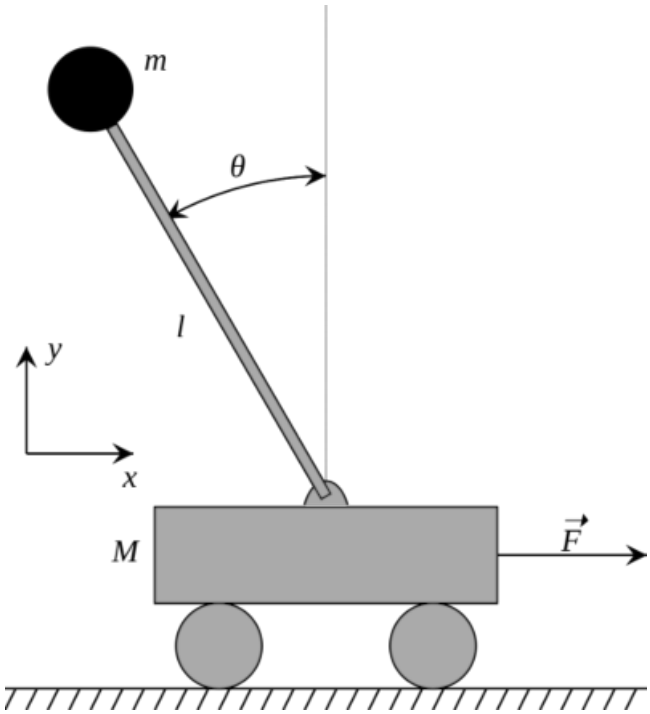

Figura 1.1: Pole balacing system

1.4 Strategie di Reinforcement Learning

La strategia che in genere consente ad un agente di massimizzare la ricompensa è quella che stabilisce un bilanciamento tra esplorazione ed exploit delle azioni, in cui col termine esplorazione si indica l'esecuzione di azioni non ancora esplorate mentre con exploit la scelta di un'azione già esplorata in base alla ricompensa media ottenuta così definibile:

$$Q_k(a) = \frac{R_1 + R_2 + ... + R_k}{k_a} \tag{1.1}$$

Tale metodo viene denominato ϵ-greedy in quanto il parametro che lo controlla è ϵ, definito dall'intervallo $0 < \epsilon < 1$: con una probabilità $(1 - \epsilon)$ verrà utilizzata la strategia ad exploit in cui l'agente sceglierà l'azione che in base all'esperienza pregressa risulta avere la ricompensa media più alta mentre con una probabilità ϵ sceglierà di attivare una fase di esplorazione eseguendo un'azione in maniera randomica. Il parametro ϵ può essere definito come un valore fisso oppure può esser fatto variare durante la fase di apprendimento diminuendo la fase esplorativa man mano che la ricompensa media ottenuta si avvicina al valore ottimale. Questo tipo di strategia si applica spesso a problemi di

tipo multi-armed bandit ossia laddove è necessario scegliere un'azione tra n possibili in cui ognuna di esse, se ripetuta nel tempo, fornisce una ricompensa media diversa.

Nella risoluzione di problemi per i quali non è opportuna la scelta casuale di un'azione ma occorre conoscere l'esatta efficacia dell'una rispetto all'altra si tende ad utilizzare una distribuzione delle probabilità per scegliere la più idonea: la funzione softmax in tal caso fornisce tale indicazione tra le diverse opzioni e ne permette una classificazione in base ad un criterio di gradimento, le peggiori avranno una probabilità piccola o vicina allo 0. Dal punto di vista matematico viene così definita:

$$Pr(A) = \frac{e^{\frac{Q_k(A)}{\tau}}}{\sum_{i=1}^{n} e^{\frac{Q_k(i)}{\tau}}} \tag{1.2}$$

La funzione ha come parametro di ingresso un vettore di coppie azioni-valore mentre τ è un parametro, definito "temperatura", che consente di scalare la distribuzione delle probabilità delle azioni. L'esatto valore di questo parametro si determina mediante sperimentazione: una quantità elevata farà in modo che le probabilità siano molto simili tra di loro mentre una bassa che vengano accentuate le differenze tra queste e quindi tra le azioni stesse.

1.5 Processi decisionali markoviani

Oltre al concetto di azione da eseguire, nella risoluzione di una determinata classe di problemi, definiti di tipo contestuale, è presente il concetto di stato: nell'apprendimento tramite reinforcement learning in tali casi si applica la proprietà di Markov che porta a selezionare le azioni da eseguire solo in base allo stato corrente in quanto il solo stato corrente contiene sufficienti informazioni per scegliere le azioni ottimali alla massimizzazione delle future ricompense. Ciò significa che le decisioni effettuate in un dato momento s_t sono le uniche ad avere effetto su s_{t+1} mentre quelle in {s0,s1,...,s_{t-1}} non avranno effetto. Ogni problema in cui è individuabile tale proprietà è definito MDP (Markov Decision Process).
Tramite il concetto di MDP possiamo definire in modo più formale il reinforcement learning come un algoritmo o agente il quale compie della azioni in un environment in cui quest'ultimo è definito come un processo che produce stati, azioni e ricompense. L'agente ha accesso solo allo stato corrente $s_t \in S$ il quale contiene tutti i dati di suo interesse relativi ad un particolare istante di tempo t. Attraverso queste informazioni esegue un'azione $a_t \in A$, che in modo deterministico o probabilistico portano l'environment in un nuovo stato

s_{t+1}. La probabilità che si verifichi il passaggio dallo stato attuale ad uno nuovo quando viene compiuta un'azione si definisce probabilità di transizione. L'agente riceve una ricompensa r_t per aver compiuto un'azione a_t in uno stato s_t che lo porterà in un nuovo stato s_{t+1}: l'obiettivo principale sarà quindi quello di massimizzare la ricompensa generata dalla transizione $s_t \rightarrow s_{t+1}$ e non dall'azione eseguita, dato che questa può in modo probabilistico portarlo anche in uno stato peggiore. In sostanza, se effettuassimo un salto da un tetto ad un altro potremmo atterrare perfettamente su quest'ultimo oppure cadere nel vuoto ma quello che importa è portarci verso i due possibili stati, non il salto in sè.

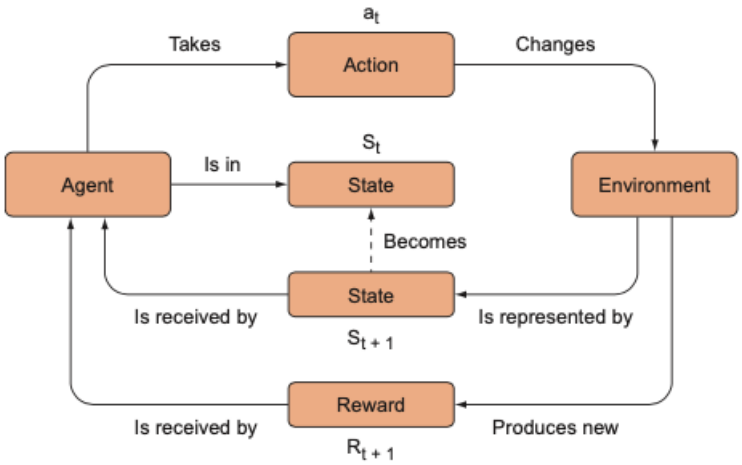

Figura 1.2: Schema generale di un algoritmo di Reinforcement Learning

1.6 Value function o policy function

Esistono due approcci fondamentali per risolvere i problemi in RL: le metodologie basate su value functions e quelle basate su policy functions. Una policy π è una funzione che mette in relazione uno stato con una distribuzione di probabilità di un set di possibili azioni che possono essere eseguite in quello stato ovvero decide la strategia di un agente in un certo environment.

$$\pi, s \rightarrow Pr(A|s), dove\ s \in S \tag{1.3}$$

La probabilità di ogni azione nella distribuzione rappresenta quella che ognuna ha di produrre la ricompensa più alta: una policy π si definisce ottima se è in grado di massimizzare la ricompensa.

$$\pi_* = argmax\ \mathbb{E}(R|\pi) \tag{1.4}$$

Ci sono due modi per determinare la policy ottima:

- Direttamente: insegnando all'agente a riconoscere quale azione è la migliore dato lo stato corrente;
- Indirettamente: insegnando all'agente quali stati portano alla maggiore ricompensa ed inducendolo ad eseguire le azioni che lo portano ad essi.

Il metodo indiretto ci porta alla definizione di value functions, funzioni che mettono in relazione uno stato o una coppia azione-stato alla ricompensa che si ottiene per essere in un determinato stato oppure per compiere un'azione in un determinato stato. Possiamo quindi definire una value function nel seguente modo:

$$V_\pi(s) = \mathbb{E}(R|s, \pi) \tag{1.5}$$

La funzione V_π è una funzione che accetta uno stato s e restituisce una ricompensa compiendo in esso delle azioni secondo una policy π. Se conosciamo una policy ottima π_* possiamo determinare anche la value function corrispondente $V_*(s)$ e viceversa:

$$V_*(s) = max\ V_\pi(s) \forall s \in S \tag{1.6}$$

Per quanto riguarda il processo inverso la policy ottima può essere determinata scegliendo tra tutte le azioni disponibili nello stato corrente s_t quella che massimizza $\mathbb{E}_{s_{t+1} \sim \mathrm{T}(s_{t+1}|s_t,a)}[V_*(s_{t+1})]$ dove $\mathrm{T}(s_{t+1}|s_t, a)$ è la transizione dinamica che mappa una coppia stato azione ad una distribuzione di stati all'istante $t + 1$. Nel caso di problemi contestuali in cui la relazione con la ricompensa coinvolga una coppia stato-azione parliamo di Q-function che possiamo così definire:

$$Q_\pi(s|a) \rightarrow \mathbb{E}(R|a, s, \pi) \tag{1.7}$$

Q_π è una funzione, simile a V_π, che data una coppia (s, a) composta da uno stato ed un'azione compiuta in esso restituisce una ricompensa in base ad una policy π. In caso di ottimalità della policy il valore di Q_* è dato da:

$$Q_*(s, a) = max\pi\ \mathbb{E}[R_t|s_t = s, a_t = a, \pi] \tag{1.8}$$

1.7 Metodi di Reinforcement Learning

Una delle caratteristiche più importanti di un algoritmo di RL riguarda la possibilità che un agente abbia accesso o debba apprendere un modello dell'environment. Con il termine modello si intende una funzione in grado di determinare delle transizioni da uno stato ad un altro e le relative ricompense. I modelli vengono utilizzati per la pianificazione intesa come modalità con cui viene scelta un'azione considerando le possibili situazioni future che da essa derivano. I metodi di Reinforcement Learning che usano modelli definiti e la pianificazione sono noti come *model based*: in essi sono note le probabilità di transizione e le funzioni di reward. Esistono, invece, metodi classificati come *model free* in cui le probabilità sono sconosciute e bisogna stimare sia la policy che la funzione valore attraverso rollout del sistema ovvero applicando uno o più step di simulazione per testare le possibili conseguenze, utilizzando cioè una strategia trial-and-error. Visto che un modello deve essere definito accuratamente per poter essere impiegato in maniera efficiente, l'approccio model-free viene utilizzato quando questo processo è di difficile implementazione: i metodi model-free possono perciò configurarsi come componenti fondamentali di quelli model-based.
Monte Carlo, Temporal Difference e Policy Search sono i metodi model-free più comunemente usati e di cui si eseguirà una breve trattazione al fine di introdurre gli algoritmi che verranno utilizzati nel seguito: Q-learning e Deep Q-learning.

1.7.1 Metodi Model-Free

I metodi model-free possono essere applicati a molti problemi di Reinforcement Learning che non richiedono alcun modello dell'ambiente. Molti approcci model-free cercano di apprendere la funzione valore e da essa inferire la policy ottimale oppure ricercano la policy ottimale direttamente nello spazio dei parametri della policy stessa: questi approcci possono essere classificati come approcci on-policy o approcci off-policy. I metodi on-policy utilizzano la policy corrente per generare azioni e la utilizzano per aggiornare la policy stessa mentre i metodi off-policy utilizzano una policy di esplorazione diversa per generare azioni rispetto alla policy che viene aggiornata.

Metodi Monte Carlo

I metodi Monte Carlo rappresentano una metodologia di risoluzione dei problemi di Reinforcement Learning basata sulla media delle ricompense ottenute. Per poter applicare questo principio in modo ottimale questi metodi vengono

utilizzati solo con task di tipo episodico ossia si assume che l'esperienza sia divisa in episodi e che tutti terminino indipendentemente dalle azioni effettuate; solo al completamento di un episodio vengono eseguite le stime dei valori ed aggiornate le policy. Un altro principio su cui si basano i metodi Monte Carlo è la GPI (Generalized Policy Iteration) che è uno schema iterativo composto da due processi: il primo prova a costruire un'approssimazione della funzione valore basandosi sulla policy corrente (policy evaluation step); il secondo migliora la policy rispetto alla funzione valore corrente (policy improvement step). Nei metodi Monte Carlo, quindi, in prima istanza per determinare la funzione valore si utilizza la tecnica del rollout eseguendo la policy corrente sul sistema e stimando la funzione valore attraverso il reward accumulato sull'intero episodio e la distribuzione degli stati incontrati; in seconda istanza, la policy corrente è migliorata attraverso tecnica greedy. Utilizzando questi due step in modo iterativo, è possibile dimostrare che l'algoritmo converge alla funzione e alla policy dal valore ottimale. Sebbene i metodi Monte Carlo siano semplici nella loro implementazione, richiedono un gran numero di iterazioni per la loro convergenza e soffrono di una grande varianza nella stima della funzione valore.

Metodi Temporal Difference

I metodi Temporal Difference (TD) sono costruiti sull'idea della GPI ma differiscono dai metodi Monte Carlo nell' evaluation step della policy: invece di usare la somma totale di reward, questi metodi calcolano l'errore temporale, ovvero la differenza tra la nuova stima e la vecchia stima della funzione valore, considerando il reward ricevuto al time step corrente e utilizzandolo per aggiornare la funzione valore. Questo tipo di aggiornamento riduce la varianza ma incrementa il bias nella stima della funzione valore. L'equazione di aggiornamento della funzione valore è data da:

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V(s') + V(s) \right] \tag{1.9}$$

dove α è il fattore di learning (learning rate), r è il reward ricevuto nell'istante corrente, s' è il nuovo stato e s è il vecchio stato. Quindi i metodi TD aggiornano la funzione valore ad ogni time step, al contrario dei metodi Monte Carlo che aspettano il completamento dell'episodio.

1.8 Q-learning

Q-learning è un algoritmo, sviluppato nel 1989 da Watkins, in cui l'action-value function che si va ad apprendere approssima direttamente Q_* 1.8, indipendentemente dalla policy che si sta utilizzando. Tutto ciò semplifica enormemente l'analisi dell'algoritmo, determinando una rapida convergenza dei tentativi. Ricorrendo alla proprietà di Markov e all'equazione di Bellman, un algoritmo di Q-learning ha la seguente forma ricorsiva:

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r_{t+1} + \gamma Q_\pi(s_{t+1}, \pi(s_{t+1}))] \tag{1.10}$$

in cui il parametro $\gamma \in [0, 1]$ è una fattore di sconto che ci permette di valorizzare maggiormente le ricompense più prossime rispetto a quelle future. Ciò significa che Q_π è soggetta a miglioramento attraverso un processo noto come bootstrap, ossia possiamo utilizzare il valore stimato per migliorare il nostro processo di stima. Queste sono le fondamenta del Q-learning e dell'algoritmo SARSA (state-action-reward-state-action):

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha\delta \tag{1.11}$$

dove α è il learning rate e $\delta = Y - Q_\pi(s_t, a_t)$ l'errore di differenza temporale (TD error), con Y il target di un problema di regressione standard. L'algoritmo SARSA, viene usato per migliorare la stima di Q_π attraverso le transizioni generate dalla policy derivata dal calcolo di Q_π stesso, per questo è di tipo on-policy. Da questo risulta che $Y = r_t + \gamma Q_\pi(s_{t+1}, a_{t+1})$. L'algoritmo di Q-learning invece è di tipo off-policy visto che Q_π si aggiorna tramite delle transizioni che non necessariamente vengono generate da una policy derivata dal suo calcolo. In questo caso il fattore Y si ottiene quindi come $Y = r_t + \gamma max\ Q_\pi(s_{t+1}, a)$ che approssima direttamente Q_*. La definizione completa di questo algoritmo è quindi la seguente:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{1.12}$$

Per determinare Q_* da un arbitrario Q_π usiamo quindi una policy generale in modo iterativo dove l'iterazione consiste nella valutazione della policy e nel suo miglioramento. La valutazione della policy migliora la stima della value function la quale viene ottenuta minimizzando gli errori di differenza temporale ottenuti dalle transizioni generate dalla policy stessa. Man mano che le stime migliorano anche la policy migliora ed è in grado di scegliere le azioni direttamente basandosi sulla value function aggiornata.

Capitolo 2

Reinforcement Learning in Python

Python è uno dei più noti linguaggi di programmazione per l'elaborazione dei dati e, pertanto, gode di una grande quantità di utili librerie aggiuntive, sviluppate dalla sua comunità. Sebbene le prestazioni dei linguaggi interpretati, come Python, per compiti intensivi dal punto di vista computazionale siano inferiori rispetto a quelle dei linguaggi di programmazione a basso livello, alcune librerie di estensione, come NumPy e SciPy, sono state sviluppate proprio basandosi su implementazioni a basso livello, in Fortran e C, in modo da accelerare le operazioni sui vettori, che devono operare su array multidimensionali. Attraverso l'utilizzo di questi strumenti verrà illustrato come implementare l'algoritmo del Q-learning di cui abbiamo parlato nel precedente capitolo. Questo algoritmo con utilizzo della forma tabellare prevede una Q-table in cui vengono rappresentate le coppie stato-azione: per poter essere implementato in tale modo, lo stato dell'ambiente, che assume valori continui nel tempo, deve essere oggetto di discretizzazione tramite bucket. Per sviluppare l'algoritmo di Q-learning, evitando di doversi preoccupare di realizzare da zero ambienti complessi, verrà utilizzato Gym OpenAI, un ambiente di simulazione contenente una collezione di environment creati per il testing e l'implementazione di algoritmi di Reinforcement Learning.

2.1 OpenAI Gym

OpenAI Gym si basa sui fondamentali del Reinforcement Learning e fornisce una struttura che rispecchia l'interazione tra agenti e ambiente: l'agente opera in un environment intraprendendo azioni e provando a massimizzare un certo *reward* che viene rilasciato a fronte delle azioni eseguite all'interno di una struttura di natura episodica. La classe principale *Env* di questa li-

breria rappresenta l'environment ed espone diversi metodi ed attributi che ne rappresentano tutte le caratteristiche, ovvero:

- una lista di azioni che possono essere eseguite, sia di tipo discreto che continuo o la combinazione di entrambe;
- l'osservazione di ciò che accade nell'environment a seguito dell'esecuzione delle azioni;
- un metodo chiamato step che esegue l'azione e restituisce l'osservazione e la ricompensa corrente nonchè l'indicazione che l'episodio si è concluso;
- un metodo chiamato reset che imposta l'environment al suo stato inziale;
- un metodo chiamato render che mostra l'agente nel nuovo stato.

Grazie ad OpenAI Gym possiamo simulare il problema del pole balancing e provare a risolverlo tramite Reinforcement Learning. Infatti, come accennato precedentemente, l'ambiente è caratterizzato da una natura episodica per cui deve essere presente uno stato terminale: ogni episodio deve concludersi sia in caso di raggiungimento dell'obiettivo sia in caso contrario. In caso l'obiettivo venga raggiunto parliamo di stato terminale di successo, altrimenti di stato terminale d'insuccesso. L'esempio di codice 2.1 crea un'istanza del Cart Pole system e ne mostra l'animazione.

Figura 2.1: Simulazione del Cart Pole in Ai Gym

```
import gym

def poleTest():
    env = gym.make("CartPole-v0")
    env.reset()
    for i_episode in range(100):
        done = False
        obs = env.reset()
        t = 0
        while not done:
            env.render()
            action = env.action_space.sample()
            obs, reward, done, info = env.step(action)
            t += 1
            if done:
                print("Done after {} steps".format(t))
                break;
    env.close()

if __name__ == '__main__':
    poleTest()
```

Listato 2.1: Esempio di come istanziare l'ambiente CartPole

Lo stato corrente del sistema è rappresentato da un vettore contenente quattro componenti $x, \dot{x}, \theta, \dot{\theta}$ che rappresentano rispettivamente: la posizione del cart, la sua velocità, l'angolo d'inclinazione del pole e la sua velocità angolare. Le azioni possono assumere due valori: left (0) e right (1), l'episodio termina se:

1. l'angolo d'inclinazione del pole è maggiore di $\pm 12°$ dall'asse verticale;
2. la posizione del cart è maggiore di ± 2.4 cm dal centro;
3. la lunghezza dell'episodio supera i 200 step.

Il reward utilizzato in questi ambienti è cumulativo: l'agente riceve un reward di 1 per ogni step eseguito (incluso quello finale) che viene sommato al precedente all'interno dello stesso episodio. Più il reward è alto più il pole viene mantenuto stabile e bilanciato. Nella versione v0 il problema si considera risolto se la media del reward calcolata su 100 episodi è maggiore o uguale a 195.

2.2 Implementazione del Q-learning in Python

In questa sezione illustreremo come implementare un algoritmo di Q-learning allo scopo di bilanciare il Cart Pole più a lungo possibile. Come illustrato nel primo capitolo l'algoritmo Q-learning utilizza l'equazione di Bellman per definire una funzione Q in grado di quantificare la ricompensa attesa ottenuta eseguendo all'istante t un'azione a_t in un dato stato s_t. La definizione matematica di tale concetto è la seguente:

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|(s_t, a_t)] \tag{2.1}$$

dove $R_i, i = t+1, ...$ è la ricompensa futura e γ il fattore di sconto. L'obiettivo è tenere traccia ed aggiornare i valori ottenuti dalle Q-function attraverso un processo iterativo volto ad esplorare tutte le possibili combinazioni di stati ed azioni ed i cui risultati verranno mantenuti in una tabella (Q-table). Il Q-learning si applica solo in presenza di un numero discreto di stati perciò, visto che nel Cart Pole questi possono assumere valori continui, si rende necessario applicare un metodo di discretizzazione attraverso una tecnica a bucket. Fissato il numero di bucket per ogni parametro dello stato si sfrutta una funzione che, preso in input lo stato in forma non discretizzata, permette lo "smistamento" e quindi la discretizzazione restituendo il bucket a cui viene assegnato. La funzione bucketize() mostrata di seguito implementa quest'operazione.

```
def bucketize_state_value(state_value):
    ''' Discretizes continuous values into fixed buckets'''
    bucket_indices = []
    for i in range(len(state_value)):
        if state_value[i] <= state_value_bounds[i][0]: # violates
            lower bound
            bucket_index = 0
        elif state_value[i] >= state_value_bounds[i][1]: # violates
            upper bound
            bucket_index = no_buckets[i] - 1 # put in the last bucket
        else:
            bound_width = state_value_bounds[i][1] -
                state_value_bounds[i][0]
            offset = (no_buckets[i]-1) * state_value_bounds[i][0] /
                bound_width
            scaling = (no_buckets[i]-1) / bound_width
            bucket_index = int(round(scaling*state_value[i]-offset))
        bucket_indices.append(bucket_index)
    return(tuple(bucket_indices))
```

Listato 2.2: Funzione di discretizzazione degli stati

Nel Cart Pole system, fissando la dimensione del bucket per lo stato a (1, 1, 6, 3), ed essendo la dimensione del vettore delle azioni pari a 2 la dimensione della Q-table risulta $1 \times 1 \times 6 \times 3 \times 2$. Il listato 2.5 mostra l'implementazione dell'algoritmo di Q-learning in cui possiamo individuare quattro step principali:

1) Selezionare un'azione attraverso una policy $\epsilon - greedy$ in cui ϵ o exploration rate determina l'attitudine dell'agente ad esplorare piuttosto che al riutilizzo dell'esperienza precedentemente accumulata (exploit). Il valore di questo parametro viene calcolato attraverso la funzione select_explore_rate la cui policy sarà quella di favorire inizialmente l'esplorazione per poi passare ad un maggior utilizzo dell'exploit col trascorrere degli episodi. Durante l'esplorazione l'azione sarà selezionata in modo random mentre durante l'exploit verrà scelta in base alle esperienze passate ossia scegliendo quella che presenta nell Q-table il valore massimo per lo stato corrente. In linguaggio matematico quest'ultima fase si rappresenta con la seguente:

$$a(s) = \arg\max_{a'} Q(s, a') \tag{2.2}$$

L'implementazione dei concetti sopra esposti è la seguente:

```
def select_explore_rate(x):
    # change the exploration rate over time.
    return max(min_learning_rate, min(1.0, 1.0 -
        math.log10((x+1)/25)))

def select_action(state_value, explore_rate):
    if random.random() < explore_rate:
        action = env.action_space.sample() # explore
    else:
        action = np.argmax(q_value_table[state_value]) # exploit
    return action
```

Listato 2.3: Selezione dell'azione e di epsilon

2) Osservare il nuovo stato ed accumulare il reward in seguito all'esecuzione dell'azione selezionata.

3) Aggiornare la Q-table utilizzando l'equazione 1.12. Il fattore α che compare in quest'equazione è noto come learning rate o tasso di apprendimento, viene decrementato in modo monotòno dal valore 1 fino ad un minimo di 0.1 durante il processo di training. Esso determina con quale estensione le nuove informazioni acquisite sovrascriveranno le vecchie informazioni. Un fattore 0 impedirebbe all'agente di apprendere, al contrario un fattore pari ad 1 farebbe

sì che l'agente si interessasse solo delle informazioni recenti. Risultati sperimentali dimostrano che il learning rate dinamico utilizzato in questo algoritmo genera risultati migliori rispetto ad un tasso di apprendimento fisso.

```
def select_learning_rate(x):
    # Change learning rate over time
    return max(min_learning_rate, min(1.0, 1.0 -
        math.log10((x+1)/25)))
```

Listato 2.4: Selezione del tasso di apprendimento

L'unico fattore fisso è γ, il discount rate, che determina l'importanza delle ricompense future. Un fattore pari a 0 farà in modo che l'agente consideri solo le ricompense attuali, mentre un fattore tendente ad 1 lo renderà attento anche a quelle che riceverà in un futuro a lungo termine. In letteratura è comune utilizzare discount rate prossimo ad 1 perciò è stato scelto $\gamma = 0.99$.

4) Aggiornare lo stato corrente e ripetere i passi da 1 a 3 nella successiva iterazione.

```
totaltime = 0
for episode_no in range(max_episodes):

    explore_rate = select_explore_rate(episode_no)
    learning_rate = select_learning_rate(episode_no)

    learning_rate_per_episode.append(learning_rate)
    explore_rate_per_episode.append(explore_rate)

    # reset the environment while starting a new episode
    observation = env.reset()

    start_state_value = bucketize_state_value(observation)
    previous_state_value = start_state_value


    done = False
    time_step = 0

    while not done:
        action = select_action(previous_state_value, explore_rate)
        observation, reward_gain, done, info = env.step(action)
        state_value = bucketize_state_value(observation)
        best_q_value = np.max(q_value_table[state_value])
```

```
        #update q_value_table
        q_value_table[previous_state_value][action] += learning_rate
            * (
            reward_gain + discount * best_q_value -
            q_value_table[previous_state_value][action])

        previous_state_value = state_value

        if episode_no % 100 == 0 and _DEBUG == True:
            print('Episode number: {}'.format(episode_no))
            print('Time step: {}'.format(time_step))
            print('Previous State Value:
                {}'.format(previous_state_value))
            print('Selected Action: {}'.format(action))
            print('Current State: {}'.format(str(state_value)))
            print('Reward Obtained: {}'.format(reward_gain))
            print('Best Q Value: {}'.format(best_q_value))
            print('Learning rate: {}'.format(learning_rate))
            print('Explore rate: {}'.format(explore_rate))

        time_step += 1
        # while loop ends here

    if time_step >= solved_time:
        no_streaks += 1
    else:
        no_streaks = 0

    if no_streaks > streak_to_end:
        print('CartPole problem is solved after {}
            episodes.'.format(episode_no))
        break

    # data log
    if episode_no % 100 == 0:
        print('Episode {} finished after {} time
            steps'.format(episode_no, time_step))
    time_per_episode.append(time_step)
    totaltime += time_step
    avgtime_per_episode.append(totaltime/(episode_no+1))
    # episode loop ends here
env.close()
```

Listato 2.5: Algoritmo di Q-learning

Nel seguente listato è presente la configurazione iniziale del sistema in cui si può vedere che il problema risulta risolto se per 120 episodi consecutivi il pole viene mantenuto in equilibrio per almeno 199 step.

```
import gym
import numpy as np
import random
import math
import matplotlib.pyplot as plt

env= gym.make('CartPole-v0')

no_buckets = (1,1,6,3)
no_actions = env.action_space.n

state_value_bounds = list(zip(env.observation_space.low,
                              env.observation_space.high))
state_value_bounds[1] = (-0.5, 0.5)
state_value_bounds[3] = (-math.radians(50), math.radians(50))

print(state_value_bounds)
print(len(state_value_bounds))
print(np.shape(state_value_bounds))
print(state_value_bounds[0][0])

action_index = len(no_buckets)

# define q_value_table
q_value_table = np.zeros(no_buckets + (no_actions,))

# user-defined parameters
min_explore_rate = 0.1
min_learning_rate = 0.1
max_episodes = 1000
max_time_steps = 250
streak_to_end = 120
solved_time = 199
discount = 0.99
no_streaks = 0
```

Listato 2.6: Configurazione iniziale dell'algoritmo

2.3 Training dell'algoritmo Q-learning

In questa sezione si evidenziano le specifiche degli esperimenti effettuati allo scopo di valutare le performances dell'algoritmo di Q-learning descritto sin qui. L'hardware di riferimento su cui ne viene avviata l'esecuzione è un I-Mac dotato di un processore Intel core i5 di nona generazione a 6 core e di 8GB di memoria RAM. Ogni episodio di training del Cart Pole ha durata massima 200 step, limite posto per non allungare ulteriormente i tempi di learning: l'episodio termina prima se il bilanciamento fallisce, altrimenti si protrae fino alla durata massima se l'oggetto risulta bilanciato. Si ricorda che il reward utilizzato in questi ambienti è cumulativo: l'agente riceve un reward di 1 per ogni step eseguito (incluso quello finale) che viene sommato al precedente all'interno dello stesso episodio. In particolare, nella versione v0 il Cart Pole si considera risolto se la media del reward calcolata su 100 episodi successivi è maggiore o uguale a 195. Allo scopo di risolvere il problema nel minor numero di episodi possibile, massimizzando quindi le sue prestazioni, sono stati eseguiti diversi test configurando i relativi parametri con valori differenti. Learning rate α, exploration rate ϵ, discount rate γ e, infine, la dimensione dei bucket per la Q-Table risultano i parametri su cui poter agire nell'algoritmo di Q-learning. La tabella successiva elenca i valori utilizzati su di essi o il range di valori assunto nel caso vengano variati durante il training dell'algoritmo:

Parametro	**Valore**
Learning rate	1.0 - 0.1
ϵ	1.0 - 0.1
discount γ	0.99
buckets	(1,1,6,3)

Tabella 2.1: Configurazione parametri Q-learning

Utilizzando questi valori l'algoritmo Q-learning risolve il problema in un range compreso tra i 300 e i 400 episodi. Il grafico seguente mostra nella parte alta l'andamento del tempo in cui il pole rimane in equilibrio al trascorrere degli episodi: la linea azzurra indica il tempo relativo ad ogni episodio, quella arancione la media su tutti gli episodi. Nella parte sottostante, invece, si evidenzia l'andamento del learning rate che decresce in monotòno al trascorrere del numero di episodi.

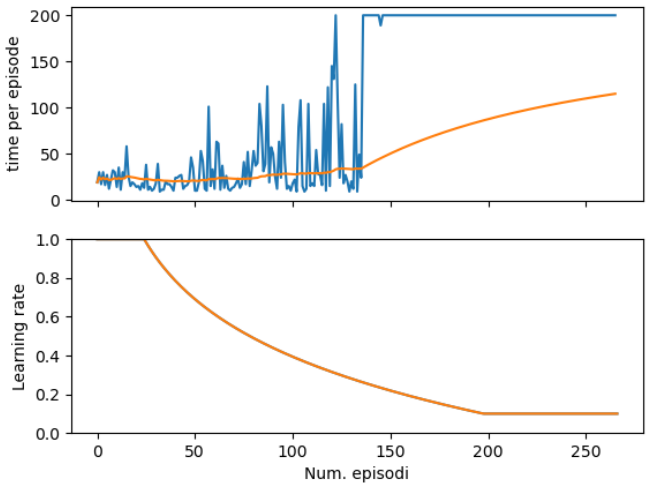

Figura 2.2: Performance dell'Algoritmo di Q-learning

La risoluzione del problema simulato e qui esposto, di natura non complessa, è stata possibile applicando all'algoritmo di Q-learning la discretizzazione degli stati che ne comporta una riduzione ai fini dell'elaborazione da parte dell'algoritmo stesso. Senza ricorrere a questo espediente che va a semplificare la gestione degli stati, pur nella sua semplicità il problema risulterebbe non affrontabile tramite Q-learning, rendendo necessaria l'introduzione di tecniche e strumenti diversi come quelli che utilizzeremo nei prossimi capitoli trattando del Deep Reinforcement Learning.

Capitolo 3

Deep Reinforcement Learning

Gli algoritmi di Deep Reinforcement Learning (DRL) hanno ricevuto, e stanno ricevendo, un notevole interesse da parte della comunità AI (Artificial Intelligence) negli ultimi anni. Con il termine Deep Reinforcement Learning ci si riferisce semplicemente all'uso di Deep Neural Networks come approssimatori di funzione per la funzione valore o la policy, in algoritmi di Reinforcement Learning. È stato dimostrato come algoritmi di DRL hanno raggiunto, e in alcuni casi superato, prestazioni a livello umano nel giocare a videogiochi Atari. Nel dettaglio verrà illustrato l'algoritmo Deep Q-learning utilizzato non solo per la risoluzione di tali videogiochi ma anche per affrontare altre problematiche di Reinforcement Learning. Per tale trattazione si rende necessario introdurre il Deep Learning quale componente basilare del DRL, in aggiunta alla componente di Reinforcement Learning già precedentemente esposta.

3.1 Deep Learning

Al Deep Learning sono attribuibili le maggiori evoluzioni nella risoluzione di problemi sui quali, prima della sua introduzione, la comunità scientifica del Machine Learning non era riuscita a progredire significativamente; ha inoltre fornito un approccio davvero efficace nell'individuare strutture complesse in dati a più dimensioni. Il deep learning permette infatti ai modelli computazionali caratterizzati da livelli multipli di processo di apprendere la rappresentazione dei dati attraverso diversi livelli di astrazione. Prima del suo avvento, il machine learning sfruttava la combinazione di accurate tecniche di feature engineering e di un'ampia esperienza del dominio applicativo per realizzare algoritmi in grado di estrapolare una rappresentazione partendo dai dati grezzi (quali, ad esempio, i pixel di un'immagine). Questa trasformazione, nota come representation learning, mirava a creare una rappresentazione dei dati più idonea ad essere elaborata dal sistema di apprendimento, quale ad esempio un

algoritmo di classificazione, il quale poteva riconoscere o classificare dei pattern da questi input. Differentemente da questo approccio il Deep Learning è quindi in grado di applicare metodi di apprendimento per rappresentazione ossia di trovare automaticamente la rappresentazione più adatta dei dati da elaborare attraverso un algoritmo di apprendimento. Questo processo viene effettuato utilizzando un'architettura contenente diversi livelli di rappresentazione che attraverso dei semplici moduli non lineari modificano i dati di partenza rappresentandoli in maniera sempre più astratta man mano che si sale di livello: ad esempio, in un problema di classificazione ciò significa che all'aumentare dell'astrazione della rappresentazione verranno esaltate le features che risultano più significative a discapito di quelle meno rilevanti. La caratteristica principale del Deep Learning è che questi livelli di rappresentazione non sono progettati da ingegneri ma si ottengono attraverso procedure di apprendimento a carattere generico. Tali livelli di rappresentazione dell'informazione si ottengono attraverso l'utilizzo di modelli basati sulle reti neurali, anch'esse disposte in livelli sovrapposti. Sebbene queste reti traggano ispirazione da concetti inerenti la neurobiologia non c'è nessuna evidenza che i meccanismi di apprendimento caratteristici del Deep Learning moderno siano coincidenti con quelli del cervello umano. Se il Machine Learning crea una relazione tra degli input e dei target attraverso l'osservazione di molti esempi di entrambi, le deep neural network sono in grado di creare tale relazione utilizzando una serie di trasformazioni che si ottengono attraverso la loro esposizione ad esempi. Tali trasformazioni vengono parametrizzate per ogni livello attraverso dei pesi (weights) e, quindi, l'apprendimento consiste nel trovare quelli appropriati per tutti i livelli della rete in modo che le relazioni tra gli input e i target siano quelle attese. Bisogna anche tenere in considerazione che i parametri di una rete possono essere milioni e la modifica di uno può influire conseguentemente sugli altri. Il controllo dell'output di una rete neurale viene eseguito attraverso l'uso di una funzione obiettivo o funzione d'errore che esplicita quanto si discosta dalle nostre aspettative: tale funzione misura la distanza tra l'output predetto dalla rete su uno specifico input e il suo target, ossia quello che vorremmo ottenere. Il Deep Learning utilizza questo errore come un segnale di feedback per variare i pesi in modo che successivamente l'errore tenda a diminuire: tale procedura, definita ottimizzazione, implementa un algoritmo che prende il nome di backpropagation, componente fondamentale del Deep Learning.

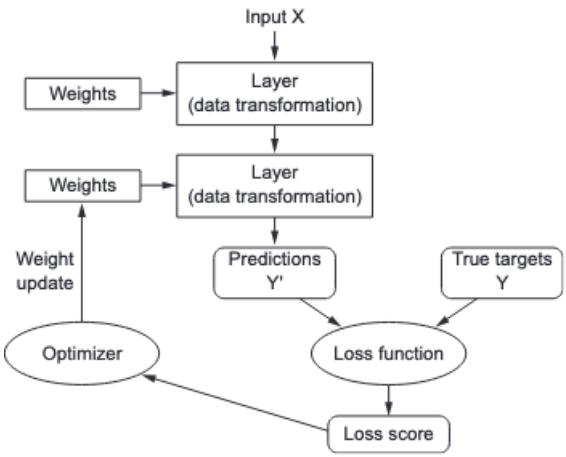

Figura 3.1: Schema di funzionamento di un processo di Deep Learning

In un processo di Deep Learning possiamo quindi evidenziare una sequenza di apprendimento in cui inizialmente ai pesi saranno assegnati dei valori casuali, quindi la rete effettuerà una serie di trasformazioni random che produrranno un errore piuttosto alto, successivamente sottoponendola a nuovi esempi questa sarà in grado di modificare il valore dei pesi in modo da portare alla riduzione dell'errore. La ripetizione di tale sequenza, definita training loop, su migliaia di esempi porterà a stabilire i valori dei pesi che minimizzano la funzione d'errore: la rete a questo punto sarà addestrata e produrrà un output non dissimile dai target. L'algoritmo di backpropagation è identificato nel processo di modifica del valore dei pesi ed è un metodo ampiamente usato per il calcolo delle derivate nelle reti neurali di tipo deep feed-forward: esso calcola, procedendo all'indietro attraverso la rete feed-forward ossia partendo dall'ultimo layer fino ad arrivare al primo, il gradiente della funzione d'errore rispetto ad ogni peso e lo utilizza per l'aggiornamento dei pesi stessi in modo da ridurre l'errore. Per calcolare il gradiente di un particolare layer è necessario considerare il gradiente di tutti i livelli successivi attraverso una catena di calcolo. Un'implementazione largamente utilizzata di questo processo è la Stochastic Gradient Descent (SGD) che utilizza un solo sottoinsieme random degli input noto come minibatch per ottenere il gradiente: questa tecnica è in grado di trovare un set ottimale di pesi molto velocemente rispetto al calcolo del gradiente ottenuto su tutto il data set. Avvalendosi infatti di tale sottoinsieme degli input vengono calcolati gli output, l'errore e la media del gradiente e si aggiornano i pesi di conseguenza; il processo si ripete utilizzando vari minibatch finchè la media dell'errore smette di decrescere.
Una volta terminata la fase di training è necessario misurare le performance del sistema utilizzando un set differente di dati in modo da stabilire se l'algo-

ritmo è in grado di generalizzare ossia dare una risposta corretta anche su dati non utilizzati durante il training.

3.1.1 Reti neurali artificiali

Come accennato in precedenza il Deep Learning si basa sulle reti neurali artificiali (Artificial Neural Network) che sono composte da entità matematiche capaci di rappresentare funzioni complesse attraverso la composizione di funzioni semplici e che per tale motivo sono uno degli strumenti attualmente più utilizzati per risolvere problemi complessi.
Sebbene i primi modelli di tali reti siano stati ispirati dalla neuroscienza e ad oggi conservino solo una parte dei meccanismi neurali presenti nel cervello umano, è però evidente che sia quelli umani sia quelli artificiali sfruttano strategie matematiche simili per approssimare funzioni complesse. Le unità fondamentali di tali reti sono rappresentate dai neuroni artificiali, costituiti da una trasformazione lineare dell'input seguita dall'applicazione di una funzione non lineare o funzione di attivazione: il neurone prende un vettore di input x e calcola la somma pesata degli input, dove il peso è indicato con w; la somma pesata viene aggiunta al bias b e passata ad una funzione di attivazione f, che produce l'output del neurone. L'equazione che rappresenta il funzionamento del neurone può essere definita come segue:

$$y_i = f(\sum_j x_j w_{i,j} + b_1) \tag{3.1}$$

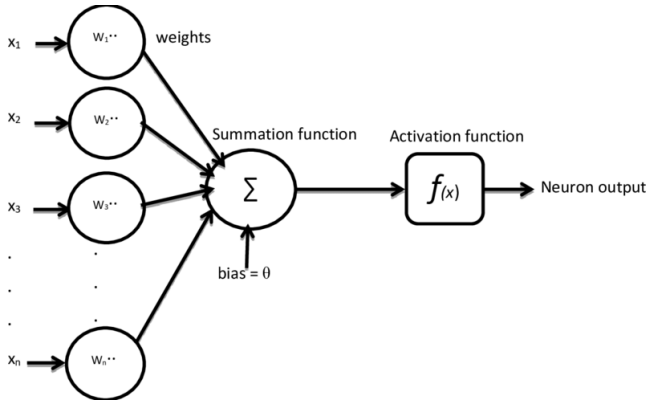

Figura 3.2: Neurone artificiale

3.1.2 Le funzioni di attivazione

La funzione di attivazione aiuta la rete ad apprendere rappresentazioni non lineari dai dati di input. Queste funzioni generalmente hanno una curva che tende a 0 o -1 per x che tende a meno infinito mentre tendono a 1 per valori di x crescenti ed hanno una pendenza costante quando x vale 0. Concettualmente lavorano molto bene perchè nella parte centrale sono più sensibili e hanno un comportamento linerare mentre agli estremi tendono ad un limite definito: questo consente di approssimare un gran numero di funzioni complesse al suo interno. Le caratteristiche di differenziabilità e continuità ne permettono il calcolo del gradiente: senza tali proprietà la rete neurale si ridurrebbe ad un modello lineare, difficile da addestrare. La figura 3.3 mostra le funzioni di attivazione maggiormente utilizzate:

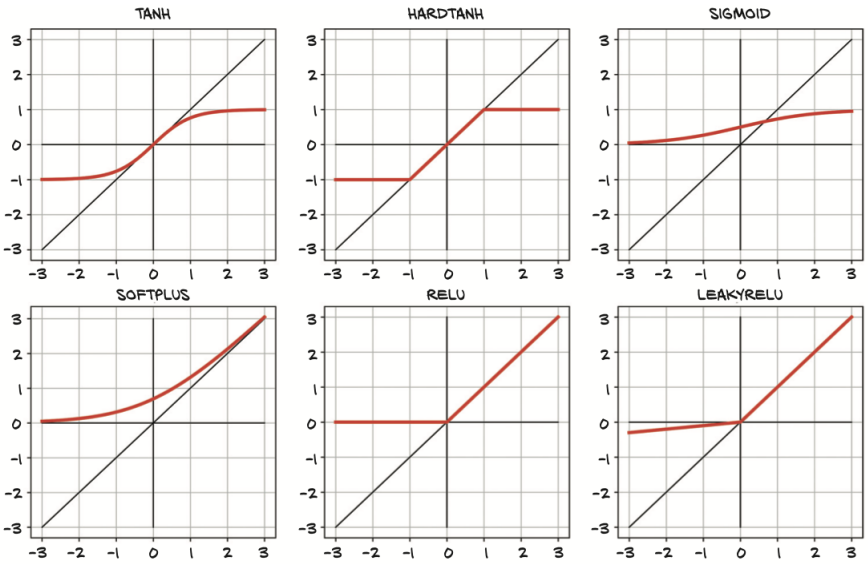

Figura 3.3: Funzioni di attivazione

Nella prima colonna sono rappresentate le funzioni di attivazioni "soft" come Tanh e Softplus mentre nella seconda le versioni "hard", Hardtanh e ReLU di cui quest'ultima è considerata una delle più performanti.

3.1.3 Feed-Forward networks

L'architettura più comunemente usata per creare la rete neurale è la Feed-Forward Network. A titolo esemplificativo, in figura 3.4 si mostra un'architettura di tipo fully connected in cui la rete neurale è totalmente interconnessa

tra vari livelli: i neuroni sono organizzati in livelli definiti layer di tre tipi ossia l'input layer, uno o più hidden layer e un output layer; il flusso di informazione parte dall'input layer, si propaga ai livelli nascosti, ed infine arriva all'output layer che computa il risultato finale. Ogni neurone nei differenti livelli applica la stessa computazione (equazione 3.1).
Nelle reti feed-forward, quindi, l'output di ogni neurone di un determinato livello è connesso a tutti gli input dei neuroni al livello successivo e non esistono cicli. I pesi dei neuroni di tali reti vengono modificati in base alla tecnica conosciuta come backpropagation di cui si è accennato in precedenza.

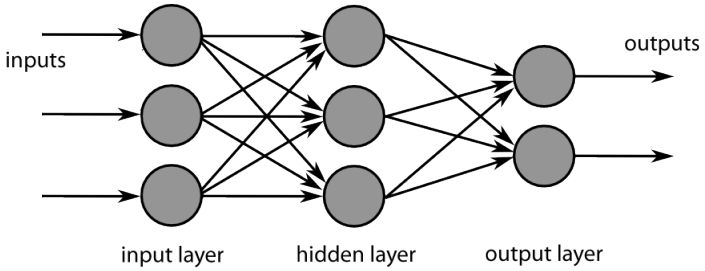

Figura 3.4: Modello di rete neurale a 3 input, 1 hidden layer e 2 output.

3.1.4 Convolutional neural networks

Oltre alle fully connected neural networks esistono altri tipi di reti neurali ad architettura Feed-Forward come ad esempio le Convolutional Neural Networks (ConvNets or CNNs), molto spesso utilizzate per affrontare problemi di classificazione e di visione artificiale. Come le tradizionali neural networks la loro struttura si ispira alle reti neuronali presenti nel cervello di uomini e animali ma le CNNs, utilizzando principi di algebra lineare quali la moltiplicazione tra matrici, sono in grado di offrire, in presenza di tali problematiche, un approccio scalabile laddove prima erano necessari complicati processi di estrazione manuale in grado di identificare un pattern all'interno di un'immagine. Si può quindi affermare, come descritto in [4], che le CNNs hanno tre principali vantaggi individuati in condivisione dei parametri, interazioni sparse e rappresentazioni equivalenti.
A differenza delle tradizionali fully connected networks, nell'analisi della struttura bidimensionale del dato di input, ad esempio un'immagine, esse utilizzano connessioni locali e pesi condivisi che rendono la rete più veloce e facile da addestrare in virtù del minor numero di parametri necessari al processo. Il modo in cui queste reti operano è simile a quello delle cellule della corteccia visiva che si dimostrano più sensibili a piccole sezioni piuttosto che a tutta la visuale agendo come dei filtri locali dell'input ed estraendo delle correlazioni spaziali

locali all'interno dei dati. Le convolutional neural networks generalmente sono composte da tre tipi di livelli diversi:

- Convolutional
- Pooling
- Fully-connected (FC)

Un tipico esempio di architettura CNN, mostrato in figura 3.5, è strutturato da un certo un numero di livelli convolutional nello stadio iniziale, seguiti da quelli di tipo pooling (downsampling) e infine, nello stadio finale, dai fully connected.

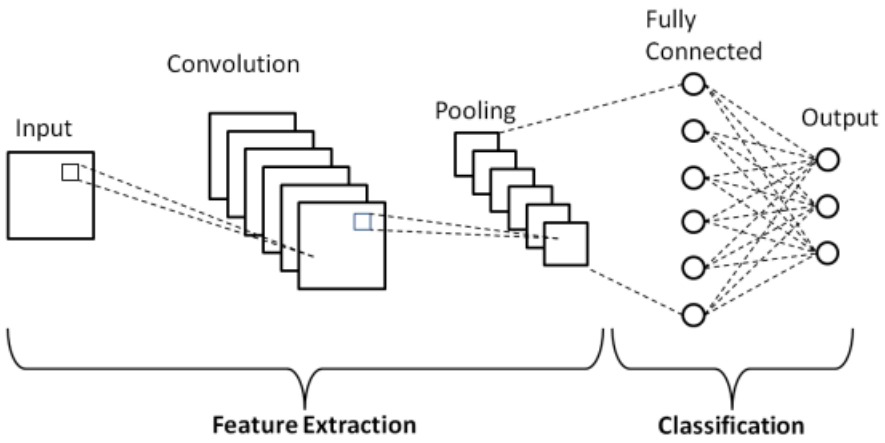

Figura 3.5: Architettura tipica di una CNN.

I livelli delle CNN utilizzano un input a 3 dimensioni $w \times h \times d$ dove w rappresenta la larghezza (width), h l'altezza (height) e d la profondità (depth): un'immagine RGB, ad esempio, avrà una profondità pari al numero dei relativi canali. Ogni livello convolutional utilizza un numero k di filtri (kernels) di dimensione $n \times n \times q$ (dove n è un numero minore di w e h mentre q può essere minore o uguale a d) per ottenere k feature map dai dati di input attraverso la convoluzione discreta. Tale operazione consiste nel far scorrere i filtri su piccole regioni del volume dell'input, nell'effettuare il prodotto scalare tra i pesi, che compongono i filtri, e la porzione di input stessa e nell'inviare il risultato di questa somma locale pesata ad una funzione non lineare quale ad esempio ReLU. Successivamente, nei livelli di tipo pooling le feature map subiscono un processo di downsampling che ha il compito di diminuire il numero dei parametri, accelerando il processo di apprendimento e controllando il fenomeno dell'overfitting. L'operazione di pooling viene applicata a regioni di dimensione $p \times p$ contigue che si trovano su tutte le feature map e può

consistere nell'applicare diversi algoritmi tra i quali l'average che effettua una media dei valori della regione oppure il maxpool che ne ricava il valore massimo. Nello stadio finale di una CNN i livelli fully connected hanno il compito di generare una rappresentazione astratta del dato e, in particolare l'ultimo livello, utilizzando ad esempio una algoritmo softmax, fornisce l'output della rete sotto forma di una lista di punteggi di classificazione ognuno dei quali rappresenta la probabilità che ha l'input di appartenere ad una determinata categoria.

3.2 Deep Q-learning

Q-learning è un algoritmo largamente usato nel Reinforcement Learning. Inizialmente era considerato un algoritmo instabile quando usato con le reti neurali e quindi il suo utilizzo veniva limitato a compiti e problemi che coinvolgevano spazi di stati a limitata dimensionalità. E' stato però dimostrato in [12] che algoritmi e tecniche di Q-Learning possono essere utilizzate con le DNN. Questo algoritmo ha mostrato il raggiungimento di performance di livello al di sopra dell'umano su sette videogames giocabili su console Atari 2600 utilizzando solamente immagini di pixel grezze come input. In questo documento ci si riferisce a tale algoritmo con il nome di Deep Q-learning o Deep Q-network (DQN). L'idea alla base del Q-learning è quella di utilizzare l'equazione di Bellman per stimare tramite aggiornamento iterativo l'action-value function, $Q_{i+1}(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q_i(s',a')|s,a]$ in modo da convergere al suo valore ottimo $Q_i \to Q_*$ se $i \to \infty$. Per molti problemi non è possibile rappresentare la Q-function come una tabella che contenga coppie di valori di s e a in quanto questa può raggiungere dimensioni molto elevate mentre invece è possibile addestrare una rete neurale a parametri θ che funga da approssimatore di funzione in grado di stimare i Q-values (ad es. $Q(s,a;\theta) \approx Q_*(s,a)$). Questo tipo di reti vengono definite col nome di Q-network e possono essere addestrate minimizzando la sequenza di funzioni d'errore $L_i(\theta_i)$ che cambiano ad ogni iterazione i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3.2}$$

$$y_i = r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \tag{3.3}$$

Nella funzione d'errore, y_i viene definito TD (temporal difference) target, $y_i - Q$ rappresenta l'errore di differenza temporale e ρ rappresenta una *behaviour distribution* ossia la distribuzione di probabilità sulle transizioni s, a, r, s'

ottenute dall'environment. Si sottolinea che i parametri provenienti dall'iterazione precedente θ_{i-1} non vengono aggiornati durante il processo di ottimizzazione della funzione d'errore $L_i(\theta_i)$: nella pratica viene usato uno snapshot dei parametri della rete relativo ad alcune iterazioni passate invece di quelli della sola ultima iterazione. Questo procedimento, dove di fatto si effettua una copia della rete e dei parametri prende il nome di target network.
Differenziando la funzione d'errore rispetto ai pesi otteniamo il seguente gradiente:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a,r,s' \sim \rho(.)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) - \nabla_{\theta_i} Q(s, a; \theta_i) \right] \tag{3.4}$$

Piuttosto che utilizzare quest'espressione del gradiente su tutti gli input è possibile sfruttare l'algoritmo SGD descritto precedentemente in modo da ridurre il carico computazionale e velocizzare il processo di ottimizzazione della funzione d'errore. Se poi a questo si integra l'algoritmo di backpropagation dell'errore è possibile addestrare la rete ripetendo il procedimento all'interno di un training loop.
L'algoritmo DQN viene ad assumere due caratteristiche: - è di tipo model-free ossia è in grado di risolvere un task di reinforcement learning utilizzando direttamente le esperienze effettuate nell'environment senza avere una sua rappresentazione concreta; - è di tipo off-policy ossia apprende attraverso una policy di tipo greedy $a = \max_a Q(a, s; \theta)$ utilizzando invece una policy differente per eseguire azioni nell'environment e acquisire dati. Quest'ultima peculiarità è solitamente una *behaviour policy* di tipo $\epsilon - greedy$ cioè seleziona con una probabilità $1 - \epsilon$ l'azione più promettente dal punto di vista della ricompensa e con probabilità ϵ un'azione in maniera randomica in modo da garantire un'adeguata esplorazione.
Un'altra tecnica chiave che viene utilizzata nelle DQN è l'experience replay in cui l'esperienza dell agente $e_t = (s_t, a_t, r_t, s_{t+1})$ viene presa ad ogni time step t e salvata in un data-set chiamato replay memory, $D = e_t, e_{t+1}, ..., e_n$, che viene popolato attraverso la ripetizione di un certo numero di episodi. In tal caso il training viene effettuato attraverso tecnica a mini-batch ovvero prelevando un sottoinsieme di campioni di esperienze estratte in modo randomico da questo replay memory. Questa tecnica permette alle esperienze passate di essere usate in più di un aggiornamento della rete, inoltre il sottoinsieme scelto in maniera casuale dalla replay memory permette di interrompere la forte correlazione presente tra esperienze successive riducendo così la varianza tra gli aggiornamenti.

Algorithm 1: Deep Q-learning con Experience Replay

Inizializza Replay Memory $\mathcal{D}$ ad una capacità $\mathcal{N}$;
Inizializza l'action-value function Q con pesi random;
for *episode* $= 1$ ***to*** M **do**
    Osserva stato iniziale s_t;
    for $t = 1$ ***to*** T **do**
        Con probabilità ϵ Seleziona un azione a_t in maniera randomica altrimenti seleziona $a_t = \max_a Q_*(s_t, a; \theta)$;
        Esegui l'azione a_t e osserva la ricompensa r_t e lo stato s_{t+1};
        Salva la tansizione (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$;
        Imposta $s_{t+1} = s_t$ Acquisisci in modo random un mini-batch di transizioni (s_t, a_t, r_t, s_{t+1}) da $\mathcal{D}$;
        Imposta

$$y_j = \begin{cases} r_j & \text{se } s_{t+1} \text{ è terminale} \\ r_j + \gamma \max_{a'} Q(s_{t+1}, a'; \theta) & \text{se } s_{t+1} \text{ è non terminale} \end{cases};$$

        Esegui uno step di gradient descendant su $(y_j - Q(s_j, a_j; \theta))^2)$
    end
end

Capitolo 4

Deep Reinforcement Learning con PyTorch

L'algoritmo di Q-learning presentato nei precedenti capitoli, la cui complessità computazionale cresce esponenzialmente con l'accrescere del numero di coppie stato-azione, soffre del problema Curse-of-Dimensionality in quanto richiede un numero di stati discreto per popolare la Q-Table. Il Deep Q-learning è in grado di risolvere questa problematica approssimando la Q-value function attraverso una rete neurale, evitando così di discretizzare gli stati con conseguente perdita di informazioni che potrebbero risultare importanti e decisive al learning dell'ambiente. In questo capitolo verrà quindi affrontato nuovamente il problema del Cart Pole ma questa volta mediante l'utilizzo del Deep Q-learning la cui implementazione si baserà sull'utilizzo di PyTorch, un framework per il Deep Learning che sta riscuotendo un notevole successo in questo ambito.

4.1 Il framework PyTorch

PyTorch è un framework semplice da apprendere, usare, estendere e farne debug: segue infatti la filosofia di Python con le sue peculiarità e best practices ma consente a chi già utilizza questo linguaggio di trovarsi subito a proprio agio.
PyTorch mette a disposizione un tipo di dato, il Tensore, in grado di contenere al suo interno numeri, vettori e matrici, e fornisce le operazioni per lavorare su di essi. Oltre a ciò, altre due caratteristiche lo rendono particolarmente adatto al Deep Learning e al calcolo scientifico in generale: da un lato fornisce un'accelerazione alla computazione utilizzando le GPU (Graphical Processing Units) e questo può significare un incremento delle prestazioni anche di 50 volte rispetto ad una normale CPU; dall'altro fornisce delle ottimizzazioni nu-

meriche sulle espressioni matematiche usate dal Deep Learning.
Per tali motivi PyTorch viene anche definito come una libreria ad alte prestazioni ottimizzata per il calcolo scientifico in Python: mentre in principio veniva utilizzata solo dai ricercatori ora si trovano sue applicazioni anche in ambiente di produzione ed è equipaggiato anche da un runtime C++ che può essere utilizzato per inferire dei modelli su dispositivi che non supportano Python.

4.1.1 I tensori e le tipologie di dato float

Le reti neurali utilizzano dati float (in virgola mobile) per gestire le informazioni. In generale, ciò significa che i dati da processare dovranno essere, in fase preliminare, convertiti in modo che le reti siano in grado di analizzarli e, a fine processo, dovranno essere restituiti in un formato di output compatibile con le esigenze del task da risolvere.
Una rete neurale ha quindi un processo di apprendimento strutturato in fasi all'interno delle quali i dati possono essere rappresentati in modo diverso ma sempre attraverso collezioni di tipi float.
PyTorch introduce rispetto a Python come struttura fondamentale di gestione dati il tensore che può essere definito anche come un'array multidimensionale, la cui dimensionalità è delineata dal numero dei suoi indici che vengono utilizzati per indirizzare un valore scalare al suo interno. I tensori contengono in genere collezioni di numeri che sono accessibili usando un indice e i cui elementi possono essere indirizzati tramite più indici. Di seguito si riporta un esempio di creazione di un tensore:

```
#In:
import torch
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

#Out:
tensor([[4., 1.],
[5., 3.],
[2., 1.]])
```

4.1.2 Tipologia degli elementi dei tensori

Mentre le collezioni di Python, liste e tuple, sono composte da oggetti allocati individualmente in memoria, i tensori di PyTorch sono invece delle viste di blocchi di memoria contigui che contengono dei tipi primitivi del C, tipicamente un float di 4-byte.
I motivi principali per cui nel Deep Learning non vengono utilizzati i tipi numerici di Python si possono così individuare:

- i numeri in Python sono oggetti quindi occuperebbero più memoria rispetto ai 32 bit allocati dal float definito da PyTorch rendendone l'utilizzo inefficiente in caso di allocazione di milioni di elementi;
- a differenza dei tensori, le liste in Python non supportano il prodotto o la somma di vettori e non consentono una ottimizzazione della distribuzione del loro contenuto in memoria;
- le liste in Python sono mono-dimensionali, seppure si possano creare liste di liste, ma questo è molto inefficiente: l'interprete Python è lento a trattarle in confronto all'utilizzo di codice compilato e ottimizzato.

Per queste ragioni le librerie per data science utilizzano NumPy o introducono strutture dedicate come i tensori di PyTorch i quali forniscono un'efficiente implementazione a basso livello delle strutture dati e relative operazioni su di esse, accessibili attraverso API di alto livello.
Il tipo di dato numerico contenuto nel tensore potrà essere definito attraverso il parametro dtype: sarà quindi possibile scegliere tra intero o float, potendo anche specificare il numero di byte per valore.

4.1.3 Storage e viste di un tensore

Come precedentemente accennato PyTorch alloca in blocchi di memoria contigui, definiti storage, i valori dei tensori: tali entità sono degli array mono-dimensionali di dati numerici. Un'istanza di un tensore è una vista dello storage capace di indirizzarne i valori usando un offset per dimensione; possono esistere più tensori che fanno riferimento ad uno stesso storage indirizzando i valori in modo diverso.
La figura 4.1 illustra questo concetto:

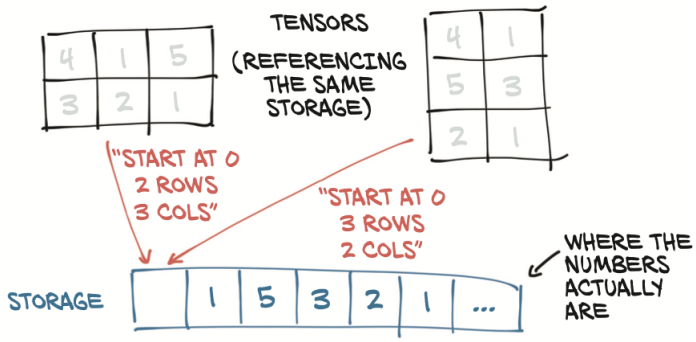

Figura 4.1: I tensori sono viste delle stesso `storage`

Possiamo creare diverse viste degli stessi dati tenendo presente la dimensione dello storage sottostante la cui memoria è allocata una sola volta. Lo storage di un tensore è accessibile usando la proprietà .storage:

```
#In:

points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points.storage()

#Out:
 4.0
 1.0
 5.0
 3.0
 2.0
 1.0

[torch.FloatStorage of size 6]
```

In quanto array mono-dimensionale, lo storage può essere indirizzato solo attraverso un indice anche se i tensori che lo rappresentano possono essere a più dimensioni.

```
#In:
points_storage = points.storage()
points_storage[0]

#Out:
4.0
```

```
#In:
points_storage[0] = 2.0

points

#Out:
tensor([[2., 1.],[5., 3.],[2., 1.]])
```

4.1.4 Metadati di un tensore

Per poter indicizzare correttamente lo storage, i tensori sono forniti di alcune proprietà che lo definiscono univocamente: shape, offset e stride. Lo shape è una tupla che indica quanti elementi per dimensione possiede un tensore; l'offset è l'indice dello storage a cui corrisponde il suo primo elemento; lo stride infine è il numero di elementi dello storage che bisogna saltare per ottenere l'elemento successivo in ogni dimensione, quindi definisce il numero di elementi per ciascuna di esse.
La relazione tra tensore e storage è in grado di rendere poco dispendiose operazioni come la trasposizione e l'estrazione di un sottotensore visto che non implicano una riallocazione di memoria ma solo l'allocazione di un nuovo tensore con differenti valori per shape, offset e stride.
La figura 4.2 illustra queste interazioni:

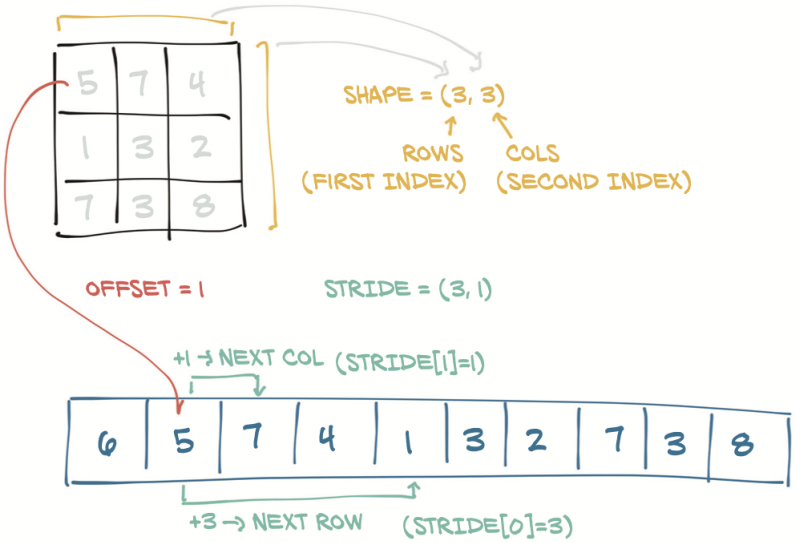

Figura 4.2: Relazione tra offset, size, e stride di un tensore. In questo caso il tensore è una vista di uno storage più ampio della sua dimensione.

4.1.5 Tensori e utilizzo di una GPU

In PyTorch i tensori possono essere processati sia su una CPU sia su una GPU (Graphics Processing Unit): a differenza della CPU, la GPU permette di aumentare la capacità computazionale e di utilizzare un parallelismo massivo. Tutte le operazione che devono essere eseguite su uno specifico tensore sono quindi reimplementate specificamente per l'utilizzo su GPU. Oltre all'attributo dtype i tensori supportano anche il concetto di device cioè il dispositivo dove vengono elaborati.
Di seguito viene riportato qualche esempio su come far elaborare un tensore alla GPU, supponendo di utilizzarne una di tipo nvidia. E' possibile creare un tensore direttamente sulla GPU:

```
#In:
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]],
device='cuda')
```

Oppure il tensore può essere copiato da una CPU alla GPU attraverso il seguente procedimento:

```
#In:
points_gpu = points.to(device='cuda')
```

In tale ultimo caso si crea una copia del tensore i cui dati saranno allocati sulla RAM della GPU piuttosto che su quella di sistema: le API che esso espone sono le medesime, rendendo più facile scrivere codice che sia indipendente dal processore utilizzato.
Nel caso in cui la macchina utilizzata disponga di più GPU è possibile allocare direttamente un tensore su una specifica GPU attraverso un intero che la identifica, in tal modo:

```
#In:
points_gpu = points.to(device='cuda:0')
```

Una volta allocato un tensore su una GPU tutte le operazioni che lo riguardano avverranno su questa, quindi, nel caso lo si voglia trasferire su una CPU, dovremo utilizzare lo stesso metodo indicando come device una CPU:

```
#In:
points_cpu = points_gpu.to(device='cpu')
```

Per completezza si riportano alcune forme più compatte con cui è possibile allocare un tensore su una GPU:

```
points_gpu = points.cuda()
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

4.2 Implementazione del Deep Q-learning applicata al Cart Pole system

L'implementazione di un algoritmo di Deep Q-learning consente di superare il problema del Curse-of-Dimensionality riscontrato nello sviluppo dell'algoritmo di Q-learning in cui, per ovviare a tale inconveniente, è stata eseguita un'operazione di discretizzazione dello stato dell'environment che ha però provocato una riduzione della quantità di informazioni ad esso relative. Il modello della rete neurale che andremo ad utilizzare per approssimare la funzione valore è di tipo feed-forward a 2 livelli con 4 nodi di input, 2 di output e un hidden layer la cui funzione di attivazione è di tipo ReLU. In questo caso, la dimensione minima dell'hidden layer verrà impostata a 30 nodi ma se ne cercherà la dimensione ottimale attraverso test incrementali. Il seguente listato mostra come definire questa rete neurale con PyTorch:

```
class QNetwork(nn.Module):
    def __init__(self, input_size, output_size, n_hidden):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, n_hidden)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(n_hidden, output_size)
        self.epsilon = Epsilon()

    def forward(self, state):
        out = self.act1(self.fc1(state))
        out = self.fc2(out)
        return out
```

Listato 4.1: Definizione della DQN attraverso PyTorch

La Deep Q Network (DQN) deve essere addestrata per poter stimare i Q-values di una determinata coppia stato-azione: per fare questo, come abbiamo spiegato nel secondo capitolo, si utilizza la funzione d'errore 3.4 dove y_i è il target Q-value $Q_*(s, a)$ per ogni iterazione definito dalla 3.3. In quest'ultima equazione r è il reward ottenuto dall'environment per la coppia stato-azione corrente e $Q(s', a'; \theta_{i-1})$ è il Q-value per lo stato successivo ottenuto usando

i pesi dell'ultima iterazione della DQN. L'algoritmo di training della rete è implementato nel metodo train_network:

```
    def train_network(env, agent, n_episode, epsilon,
        dqn_parameters, trainId, verbose=False, polyak_avg=False):
        """
        Deep Q-Learning training algorithm using double DQN, with
            experience replay
        @parma agent: instance of DQNAgent class
        @param env: Gym environment
        @param n_episode: number of episodes
        @param epsilon: object for epsilon_greedy
        """
        scores = []
        avg100Scores = []
        avgScores = []
        last100Scores = deque(maxlen=100)
        memory = deque(maxlen=dqn_parameters.mem_size)
        average100Score = 0
        solved = False
        for episode in range(n_episode):
            total_reward_episode = 0

            if polyak_avg:
                agent.soft_copy_target(0.1)
            elif episode % dqn_parameters.target_update == 0:
                agent.copy_target()

            policy = agent.gen_epsilon_greedy_policy(epsilon.value,
                dqn_parameters.action_size)
            state = env.reset()
            is_done = False

            while not is_done:

                action = policy(state)
                next_state, reward, is_done, _ = env.step(action)

                total_reward_episode += reward

                memory.append((state, action, next_state, reward,
                    is_done))

                if is_done:
```

```
                scores.append(total_reward_episode)
                avgScores.append(np.mean(scores))
                last100Scores.append(total_reward_episode)
                avg100Scores.append(np.mean(last100Scores))
                break

            agent.replay(memory, dqn_parameters.replay_size,
                dqn_parameters.gamma)

            state = next_state

        epsilon.update()

        average100Score = np.mean(scores[-min(100, len(scores)):])

        if verbose:
            print("episode: {}, score: {}, memory length: {},
                epsilon: {}, average score: {}" \
                  .format(episode, total_reward_episode,
                      len(memory), epsilon.value, average100Score))

        if average100Score >= (env.spec.max_episode_steps - 5):
            print('Training Phase: problem is solved in {}
                episodes.'.format(episode))
            solved = True
            agent.save_model(trainId, dqn_parameters.data_path)
            break

    if not solved:
        print('Training Phase: last 100 average Score:
            {}'.format(average100Score))

    printScores(scores, avgScores, avg100Scores, trainId)
```

Listato 4.2: Algoritmo di training della DQN

Il metodo train_network riceve in ingresso i seguenti parametri:

- env: istanza dell'environment del cart pole di Open Ai
- agent: istanza della classe *DQNAgent* che rappresenta l'agente
- n_episode: numero di episodi su cui verrà effettuato il training
- epsilon: istanza della classe *Epsilon* che gestisce il carattere esplorativo dell'agente

- dqn_parameters: istanza della classe *DQNParameters* che contiene i parametri caratteristici dell'algoritmo
- trainId: identificativo di un processo di training
- verbose: booleano che attiva una modalità di debug dell'algoritmo tramite stampa dello stato dell'agente e dell'environment in console
- polyak_avg: booleano che abilita la copia dei parametri della rete neurale tramite algoritmo Polyak

Prima della sua esecuzione si configura quindi una fase iniziale in cui vengono inizializzati tutti i parametri caratteristici dell'algoritmo come mostrato nel listato 4.6. La classe *DQNParameters* è quella che li contiene in maggioranza e la sua implementazione si trova nel seguente listato:

```
class DQNParameters():
    def __init__(self, action_size, lr=1e-3, mem_size=30000,
        replay_size=20, gamma=0.9, n_hidden=30, target_update=30):
        """
        @param action_size: number of actions
        @param lr: learning rate
        @param mem_size: size of replay memory
        @param replay_size: number of samples we use to update the
            model each time
        @param target_update: number of episodes before updating the
            target network
        @param gamma: the discount factor
        @param n_hidden: number of neurons of hidden layer
        @parma target_update: update interval of target network
        """
        super(DQNParameters, self).__init__()
        self.learning_rate = lr
        self.action_size = action_size
        self.mem_size = mem_size
        self.replay_size = replay_size
        self.gamma = gamma
        self.n_hidden = n_hidden
        self.target_update = target_update
```

Listato 4.3: Configurazione dei parametri caratteristici dell'algoritmo

All'interno del metodo train_network viene eseguito un training loop su un determinato numero di episodi: in ciascuno di essi il sistema esegue degli step in cui la rete produce una stima della funzione valore per ogni coppia

stato-azione. Gli errori calcolati dalla loss function saranno soggetti a backpropagation nella rete mediante un passo backward, seguendo la logica di discesa del gradiente con l'intento di minimizzare l'errore.
Similmente a quanto detto per Q-Learning, anche DQN utilizza una metodologia off-policy per bilanciare la componente esplorativa con quella di exploit nella scelta dell'azione da eseguire. La sua implementazione si trova nel metodo gen_epsilon_greedy_policy della classe *DQNAgent* 4.5, il valore di di ϵ viene gestito in modo dinamico dalla classe *Epsilon* 4.4 in cui l'exploration rate viene ridotto progressivamente durante la fase di training attraverso il parametro EPS_DECAY fino ad un minimo di 0.01.

```
EPS_START = 1.0
EPS_END = 0.01
EPS_DECAY = 0.99

class Epsilon():
    def __init__(self):
        super(Epsilon, self).__init__()
        self.value = EPS_START

    def update(self):
        self.value = max(self.value * EPS_DECAY, EPS_END)
```

Listato 4.4: Definizione dell'exploration rate

Purtroppo senza l'ausilio di tecniche come l'Experience Replay e il Double DQN il processo di apprendimento della rete neurale soffre di una certa instabilità, si è reso quindi necessario introdurne l'implementazione per stabilizzare il training. La classe *DQNAgent* mostrata nel listato 4.5 contiene la definizione dell'agente e l'implementazione di queste tecniche che vengono illustrate in modo dettagliato nei paragrafi successivi.

4.2.1 Experience replay

La tecnica dell'experience replay ci consente di risolvere un problema molto importante associato al training online degli algoritmi basato su gradient descendant: il *catastrophic forgetting*. L'essenza di questa problematica consiste nel fornire alla rete coppie stato-azione simili da cui si ottengono risultati divergenti che, se propagati ad ogni step tramite backpropagation, provocano l'impossibilità di apprendere. L'aggiornamento dei pesi, piuttosto che avvenire in modo incrementale dopo avere processato una sola esperienza, tramite l'experience replay basato sul concetto di batch update, ha luogo dopo aver

processato un certo numero di esperienze. Utilizzando questo concetto quindi si crea una memoria delle esperienze passate da cui estrarre in modo random un pacchetto o batch che è utilizzato per il training della Deep Q-network: in tal modo il processo di learning risolve il catastrophic forgetting, risultando stabile. Ogni esperienza immagazzinata è costituita da una tupla di dimensione fissa (s, a, r, s') in modo da contenere lo stato e l'azione corrente, il reward e lo stato successivo ottenuto dopo ogni iterazione. L'implementazione di questi concetti viene illustrata nella parte iniziale del metodo replay nel listato 4.5 in cui dall'oggetto replay viene estratto un mini-batch di queste tuple tramite il quale vengono popolati i valori e, per ognuna di queste, viene calcolato il Q value; dopo aver processato tutto il batch viene eseguito il metodo update che provoca l'aggiornamento dei pesi.

4.2.2 Double DQN

Nell'equazione 3.3 viene calcolato il massimo di un Q value stimato che poi si utilizzerà come target per il training della DQN. Tale pratica comporta un risvolto negativo ossia l'introduzione di una massimizzazione del bias durante l'apprendimento: dato che il Q-learning si basa sulla tecnica del bootstrap, che esegue stime partendo da stime precedenti, ciò produce nel tempo una sovrastima dei valori. Come ampiamente descritto in [3] questo problema può essere risolto attraverso una tecnica nota come *Double Q-learning* che consiste nel creare due copie della DQN, Q e Q' con quest'ultima generalmente definita *target network*. Ad intervalli di episodi regolari viene effettuate una copia dei pesi da Q a Q' mentre il backpropagation step viene effettuato solo su Q. Il valore target utilizzato per il training della rete si ottiene quindi dalla seguente formula utilizzando Q per la selezione dell'azione e Q' per la sua valutazione:

$$Q_*(s,a) \approx r_t + \gamma\, Q'(s_{t+1}, \arg\max_a Q(s_t, a_t)) \tag{4.1}$$

La copia dei pesi dalla rete Q alla rete Q' può essere eseguita anche in maniera *soft* tramite il metodo Polyak, utilizzando la seguente formula in cui θ e θ' sono rispettivamente i pesi di di Q e Q' con $0 < \tau < 1$:

$$\theta' = \tau\theta + (1-\tau)\theta' \tag{4.2}$$

L'implementazione di queste tecniche si trova nel successivo listato, all'interno dei metodi copy_target e soft_copy_target:

```
class DQNAgent():
    def __init__(self, input_size, output_size, dqn_parameter):
        self.criterion = torch.nn.MSELoss()
```

```
        self.model = QNetwork(input_size, output_size,
            dqn_parameter.n_hidden)

        self.model_target = copy.deepcopy(self.model)

        self.optimizer =
            torch.optim.Adam(self.model.parameters(),
            dqn_parameter.learning_rate)

    def update(self, s, y):
        """
        Update the weights of the DQN given a training sample
        @param s: state
        @param y: target value
        """
        y_pred = self.model(torch.Tensor(s))
        loss = self.criterion(y_pred, Variable(torch.Tensor(y)))
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()


    def predict(self, s):
        """
        Compute the Q values of the state for all actions using
            the learning model
        @param s: input state
        @return: Q values of the state for all actions
        """
        with torch.no_grad():
            return self.model(torch.Tensor(s))

    def target_predict(self, s):
        """
        Compute the Q values of the state for all actions using
            the target network
        @param s: input state
        @return: targeted Q values of the state for all actions
        """
        with torch.no_grad():
            return self.model_target(torch.Tensor(s))

    def replay(self, memory, replay_size, gamma):
        """
```

```
        Experience replay with target network
        @param memory: a list of experience
        @param replay_size: the number of samples we use to
            update the model each time
        @param gamma: the discount factor
        """
        if len(memory) >= replay_size:
            replay_data = random.sample(memory, replay_size)

            states = []
            td_targets = []
            for state, action, next_state, reward, is_done in
                replay_data:
                states.append(state)
                q_values = self.predict(state).tolist()
                if is_done:
                    q_values[action] = reward
                else:
                    q_values_next =
                        self.target_predict(next_state).detach()

                    q_values[action] = reward + gamma *
                        torch.max(q_values_next).item()

                td_targets.append(q_values)

            self.update(states, td_targets)

    def soft_copy_target(self, pa_tau=0.1):
        model_params = self.model.named_parameters()
        target_params = self.target_model.named_parameters()

        dict_target_params = dict(target_params)

        for name1, param1 in model_params:
            if name1 in dict_target_params:
                dict_target_params[name1].data.copy_(
                    pa_tau * param1.data + (1 - pa_tau) *
                        dict_target_params[name1].data)

        self.model_target.load_state_dict(dict_target_params)

    def copy_target(self):
        self.model_target.load_state_dict(self.model.state_dict())
```

```
    def gen_epsilon_greedy_policy(self, epsilon, n_action):
        """
        Epsilon greedy policy for action selection
        @param epsilon: epsilon variable
        @param n_action: total number of actions available
        """
        def policy_function(state):
            if random.random() < epsilon:
                return random.randint(0, n_action - 1)
            else:
                q_values = self.predict(state)
                return torch.argmax(q_values).item()

        return policy_function

    def gen_greedy_policy(self):
        """
        Epsilon greedy policy for action selection
        @param epsilon: epsilon variable
        @param n_action: total number of actions available
        """
        def policy_function(state):
            q_values = self.predict(state)
            return torch.argmax(q_values).item()

        return policy_function

    def save_model(self, id, data_path):
        torch.save(self.model.state_dict(), data_path +
            '/dqn_model_' + id + ".pt")

    def load_model(self, id, data_path):
        self.model.load_state_dict(torch.load(data_path +
            '/dqn_model_' + id + ".pt"))
```

Listato 4.5: Definizione dell'agente

4.3 Training dell'algoritmo Deep Q-learning

L'hardware utilizzato per il training della DQN è lo stesso utilizzato per addestrare l'algoritmo di Q-learning illustrato nel precedente capitolo, in que-

sto modo sarà possibile effettuare dei confronti sul numero di episodi necessari a ciascun algoritmo per risolvere il problema del Cart Pole. Oltre a learning rate, exploration rate e discount rate, caratteristici del Q-learning, in questo caso sono stati configurati ulteriori parametri specifici delle DQN e di questo specifico algoritmo: numero di hidden layer, numero di neuroni per ogni hidden layer, dimensione del batch di aggiornamento, frequenza di aggiornamento della target network e capacità della replay memory. Considerato che l'obiettivo era raggiungere il reward massimo eseguendo il minor numero possibile di episodi e data la semplicità del problema, la rete neurale è stata configurata con un unico hidden layer. Nella seguente tabella si riassumono i parametri ed i rispettivi valori con cui sono stati eseguiti i test:

Parametro	**Valore**
Learning rate	1.0 - 0.1
ϵ	1.0 - 0.1
discount γ	0.99
ϵ decay	0.9
Num. hidden neurons	30, 40, 50
Replay memory size	30000
Batch size	20, 25
Target network update interval	30, 35

Tabella 4.1: Parametri e valori utilizzati nel training della DQN

Implementando un grid training come illustrato nel listato 4.6 il seguente setup ha consentito di risolvere il problema in 276 episodi ed è quindi stato identificato come ottimale:

Parametro	**Valore**
Num. hidden neurons	50
Batch size	25
Target network update interval	35

Tabella 4.2: Setup ottimale dei parametri

```
def grid_train():
    env = gym.envs.make("CartPole-v0")
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n
```

```
    n_episode = 600
    n_hidden_options = [30, 40, 50]
    lr_options = [0.001]
    replay_size_options = [20, 25]
    target_update_options = [30, 35]

    for n_hidden in n_hidden_options:
        for lr in lr_options:
            for replay_size in replay_size_options:
                for target_update in target_update_options:
                    trainId = str(n_hidden) + "_" + str(lr) + "_"
                        + str(replay_size) + "_" +
                        str(target_update)
                    print("Net params: hidden layer: {} , learning
                        rate: {}, replay size {}, target update {}"\
                          .format(n_hidden, lr, replay_size,
                              target_update))
                    env.seed(1)
                    random.seed(1)
                    torch.manual_seed(1)
                    epsilon = Epsilon()
                    dqn_param = DQNParameters(action_size, \
                                        lr=lr, \
                                        mem_size=30000, \
                                        replay_size=replay_size,\
                                        gamma=0.9,
                                            n_hidden=n_hidden,\
                                        target_update=target_update)
                    agent = DQNAgent(state_size, action_size,
                        dqn_param, epsilon)
                    train_network(env, agent, n_episode, epsilon,
                        dqn_param, trainId)
    env.close()

def test_model(model_id, n_episode, verbose):
    env = gym.envs.make("CartPole-v0")
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    dqn_param = DQNParameters(action_size, n_hidden=40)
    agent = DQNAgent(state_size, action_size, dqn_param)
    agent.load_model(model_id, dqn_param.data_path)
```

```
scores = []
avg100Scores = []
avgScores = []
last100Scores = deque(maxlen=100)
average100Score = 0
solved = False
for episode in range(n_episode):
    total_reward_episode = 0

    policy = agent.gen_greedy_policy()
    state = env.reset()
    is_done = False

    while not is_done:

        action = policy(state)
        next_state, reward, is_done, _ = env.step(action)

        total_reward_episode += reward

        if is_done:
            scores.append(total_reward_episode)
            avgScores.append(np.mean(scores))
            last100Scores.append(total_reward_episode)
            avg100Scores.append(np.mean(last100Scores))
            break

        state = next_state

    average100Score = np.mean(scores[-min(100, len(scores)):])

    if verbose:
        print("episode: {}, score: {}, average score: {}" \
              .format(episode, total_reward_episode,
                  average100Score))

if average100Score >= (env.spec.max_episode_steps - 5):
    print('Test Phase: problem is solved in {}
        episodes.'.format(episode))
    solved = True

if not solved:
    print('Test Phase: last 100 average Score:
        {}'.format(average100Score))
```

```
        printScores(scores, avgScores, avg100Scores, "test_model")

        env.close()

    if __name__ == '__main__':
        test = False
        if not test:
            grid_train()
        else:
            model_id = "50_0.001_25_35"
            test_model(model_id, 100, True)
```

Listato 4.6: Grid traininig dell'algoritmo Deep Q-Network

A titolo esemplificativo l'algoritmo di training salva il modello e i suoi parametri una volta raggiunto il reward cumulativo massimo su 100 episodi, tale modello verrà caricato in memoria nella successiva fase di test in modo da verificare che il training sia avvenuto correttamente.
Il grafico seguente mostra l'andamento del reward al trascorrere degli episodi: la linea azzurra indica il reward accumulato in ogni episodio, quella arancione la media su tutti gli episodi, quella viola la media degli ultimi 100 episodi.

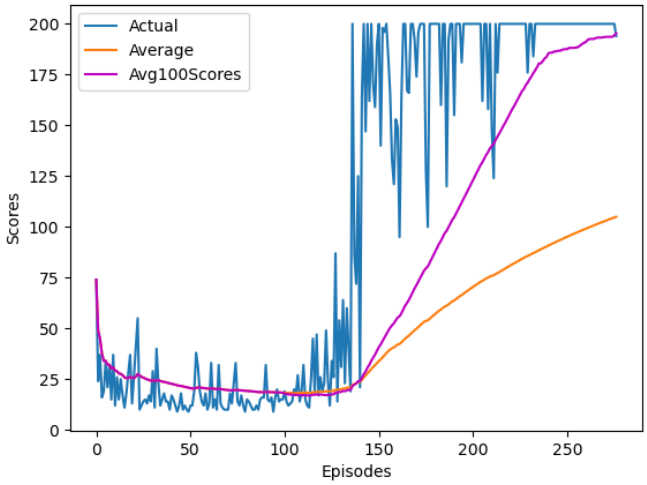

Figura 4.3: Performance dell'Algoritmo di Deep Q-learning

Data la semplicità del problema trattato, il numero di episodi necessari alla risoluzione del problema non risulta in questo caso di molto inferiore a quello riscontrato utilizzando il Q-learning. In caso di problemi più complessi, però,

questa differenza sarebbe di sicuro più marcata oppure il Q-learning potrebbe rivelarsi addirittura inapplicabile. Come ulteriore verifica della correttezza del training dell'algoritmo Deep Q-learning, per verificare che l'algoritmo stesso non si sia adattato alle casistiche incontrate in fase di addestramento, è stato sviluppato un algoritmo di test che applica il modello appena addestrato al problema del Cart pole. Se l'operazione è avvenuta correttamente il modello sarà in grado di risolvere il task ottenendo il massimo dei risultati, ciò significa che per ogni episodio il reward cumulativo sarà uguale a 200, limite impostato nell'ambiente: a differenza dell'algoritmo di training in cui la strategia utilizzata per la selezione dell'azione era di tipo epsilon greedy, qui è solo di tipo greedy in quanto l'algoritmo è già in grado di determinare qual è l'azione che determina il reward più alto. Con riferimento quindi al listato 4.6 l'algoritmo di test è implementato nel metodo test_model in cui il modello addestrato viene caricato in memoria attraverso il metodo load_model della classe *DQNAgent*, tale modello potrebbe essere inoltre utilizzato per risolvere altri task con le medesime caratteristiche senza bisogno di un ulteriore training.

Conclusioni

Fin dalla nascita dell'intelligenza artificiale, negli anni '50, i ricercatori in AI, Machine learning, scienze cognitive e neuroscienze hanno cercato di realizzare sistemi in grado di apprendere e pensare come gli esseri umani. Il Deep reinforcement learning ha permesso grossi passi in avanti nella creazione di sistemi dotati di artificial general intelligence (AGI) che possono apprendere e interagire in un ambiente. Nell'esplorazione che si è effettuata, volta ad analizzarne i principi chiave, si è dimostrato come grazie all'utilizzo degli algoritmi Q-Learning e Deep Q-learning un agente è in grado di apprendere in modo autonomo come risolvere task di controllo. Sia Q-Learning attraverso materializzazione della tabella di coppie stato-azione, che DQN tramite utilizzo di una rete neurale, ottengono ottimi risultati in termini di accuratezza nella risoluzione di un problema semplice quale il Cart Pole. L'utilizzo del toolkit openAI Gym ci consente di applicare gli algoritmi di DRL a questo problema senza dover progettare un ambiente di simulazione: mediante l'interfaccia fornita è infatti possibile realizzare ambienti standard che possono facilmente essere estesi in numerosità e quindi essere testati dagli algoritmi senza la modifica di questi ultimi.
L'algoritmo Deep Q-learning applicato a problematiche complesse può avere problemi nella fase di training ma può essere reso più stabile e performante attraverso ulteriori tecniche quali il *Dueling DQN* e il *Prioritized Experience Replay*. Il Dueling DQN scompone il Q-value come somma di V(s), il valore ottenuto nel trovarsi in un determinato stato e A(s, a), il vantaggio nell'eseguire un'azione in quello stato: valutando separatamente queste due componenti questa tecnica consente di capire quali sono gli stati migliori senza bisogno di apprezzare il risultato ottenuto dalla combinazione di ogni possibile stato e azione.
Il Prioritize Experience Replay rappresenta un'ulteriore evoluzione dell'Experience Replay basato sulla considerazione che alcune esperienze, seppur accadano in minor frequenza, possono risultare più importanti di altre durante l'apprendimento. Questo approccio quindi attribuisce alle esperienze un diverso grado di priorità in modo che durante l'apprendimento vengano riutilizzate maggiormente quelle ritenute più importanti.

L'algoritmo di Deep Q-learning che si è analizzato in questa tesi non è altro che uno degli strumenti di apprendimento messi in atto col Deep Reinforcement learning, ma non è il solo: algoritmi come Policy Gradient e Actor Critic vanno a completarne lo stato dell'arte. Policy Gradient rappresenta una classe di algoritmi che ci permette di approssimare la policy function $\pi(s)$ invece della value function $V(\pi)$ o della Q function: la rete viene quindi addestrata per restituire in output le probabilità associate alle azioni anzichè i valori delle stesse e, per tale motivo, questi algoritmi offrono alcuni vantaggi su quelli a predizione del valore come il Deep Q-learning. Innanzitutto non è necessario impostare una strategia di selezione delle azioni come quella $\epsilon - greedy$ di cui si è trattato in quanto, tramite la policy, è possibile selezionare direttamente un'azione; in secondo luogo una policy network tende a semplificare la messa a punto dell'algoritmo che non ha più bisogno, per stabilizzare il processo di apprendimento della DQN, delle tecniche di experience replay e target networks. Actor critic è un tipo di algoritmo che, unendo i vantaggi di value e policy function, cerca di apprendere una value function attraverso una policy implicita. Il componente Actor (policy) riceve lo stato dall'environment e sceglie quale azione compiere, allo stesso tempo il componente Critic (value function) riceve lo stato e il reward dalla precedente interazione e attraverso l'errore Temporal difference calcolato da questa informazione aggiorna se stesso e l'actor. Utilizzando quindi l'algoritmo policy gradient per la scelta dell'azione, Actor Critic non necessita di una policy function separata come $\epsilon - greedy$, ed è inoltre più direttamente connesso al concetto di rinforzo essendo in grado di rafforzare positivamente le azioni che determinano un reward positivo e viceversa. Daltronde la value function viene utilizzata per ridurre la varianza dei rewards utilizzati per il training della policy.

L'applicazione del DRL alla risoluzione dei giochi ATARI e ad altri task del mondo reale ha dato ottimi risultati ma il tempo di apprendimento di questi algoritmi rimane ancora troppo lento per problemi complessi in comparazione alla naturale capacità di un essere umano: 15 minuti per un uomo possono corrispondere a 115 ore di apprendimento per una DDQN. Servono ancora molti passi in avanti per raggiungere quindi livelli competitivi, alcuni se ne stanno già compiendo attraverso lo sviluppo di tecniche volte alla realizzazione di modelli pre-addestrati con conoscenze generiche che si configurano come step di partenza per la formazione di modelli più specifici, quindi a più rapido apprendimento. Ad esempio, grazie all'introduzione dello One-Shot Imitation Learning è ora possibile un apprendimento supervisionato utilizzando solo un ristretto numero di elementi in grado di descrivere un task ma capaci di creare una conoscenza generica adattabile anche a situazioni non previste generate dello stesso. L'apprendimento viene gestito in modo da raggiungere un obiet-

tivo più generico trasferendo poi questa conoscenza per risolvere task analoghi attraverso il transfer learning. Questa tecnica, usando modelli pre-addestrati come punto di partenza per la definizione di nuovi modelli, ha uno speed-up della fase di training che si trasforma quindi in una fase di tuning in cui il nuovo modello viene raffinato in base all'input fornito e all'output richiesto. Un'altra metodologia in grado di rendere l'apprendimento più rapido è il Meta-Learning che fa uso di un modello agnostico ottenuto addestrando un agente su un ampio numero di task differenti in modo che possa apprenderne una rappresentazione delle caratteristiche comuni che viene poi trasferita ad un nuovo modello attraverso un processo d'inizializzazione dei suoi parametri (weights transfer) in modo che a questo risulti necessaria solo una fase di tuning, che richiede quindi un training rapido, evitando il fenomeno dell'overfitting che può manifestarsi utilizzando un ristretto set di dati per l'addestramento.

Un altro settore di ricerca è rappresentato dal multi-agent reinforcement learning (MARL), approccio che utilizza più agenti all'interno dello stesso environment volto ad esplorare le policy multi agente che possono agire allo stesso tempo in modo alternato o simultaneo. MARL è una tecnica molto promettente ed ha riscosso un notevole successo applicata a giochi di strategia o nelle comunicazioni tra differenti veicoli autonomi consentendone un continuo avanzamento tecnologico grazie alla capacità di indagare l'intelligenza condivisa, generare ambienti più dinamici per ogni agente e applicare innovazioni all'agente stesso.

Non solo videogiochi o sistemi autonomi di apprendimento, diverse sono le applicazioni del DRL nell'industria che sono distinguibili in tre macrocategorie: la prima comprende applicazioni di controllo per la robotica, automazione nell'industria e Smart grids; la seconda riguarda le ottimizzazioni e quindi si applica alla Supply chain o al Demand forecasting; la terza infine si occupa di monitoraggi e manutenzioni quali il controllo qualità, la ricerca dei guasti e la manutenzione preventiva. In ognuna delle applicazioni menzionate il ciclo di vita del DRL, prima di raggiungere la fase finale di deployment in cui il modello addestrato viene applicato direttamente alla problematica reale e su di essa ulteriormente tarato, si compone di una fase di training atta ad eseguire una veloce simulazione ed una fase di simulazione ad alta fedeltà in cui il modello viene ulteriormente addestrato fino al raggiungimento dell'accuratezza desiderata.

Bibliografia

[1] Zai Alexander and Braw Brandon. *Deep Reinforcement Learning in action*. Manning, 2020.

[2] Stevens Eli, Antiga Luca, and Viehmann Thomas. *Deep Learning with Pytorch*. Manning, 2020.

[3] van Hasselt Hado, Guez Arthur, and Silver David. Deep reinforcement learning with double q-learning, google deepmind. *arXiv*, 2015.

[4] Goodfellow Ian, Bengio Yoshua, and Courville Aaron. *Deep Learning, Vol. 1*. MIT Press, 2016.

[5] Arulkumaran Kai, Deisenroth Marc Peter, Brundage Miles, and Bharath Anil Anthony. Deep reinforcement learning: A brief survey. *IEEE*, 2017.

[6] Yuxi Li. Deep reinforcement learning: An overview. *https://arxiv.org/abs/ 1810.06339*, 2018.

[7] Yuxi Liu. *PyTorch 1.x Reinforcement Learning Cookbook*. Packt, 2019.

[8] Lapan Maxim. *Deep Reinforcement Learning Hands-On*. Packt Publishing Ltd, 2018.

[9] Morales Miguel. *Grokking Deep Reinforcement Learning*. Manning, 2020.

[10] Sutton Richard S. and Barto Andrew G. *Reinforcement Learning: An Introduction*. The MIT Press Cambridge, Massachusetts, 2018, 2020.

[11] P. Simon. *Too Big to Ignore: The Business Case for Big Data*. Wiley, 2013.

[12] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, and Riedmiller M. Playing atari with deep reinforcement learnin. *DeepMind*, 2013.

[13] LeCun Yann, Bengio Yoshua, and Geoffrey Hinton. Deep learning. *Nature*, 2015.

[14] Fenjiro Youssef and Benbrahim Houda. Deep reinforcement learning overview of the state of the art. *Journal of Automation, Mobile Robotics & Intelligent Systems*, 2018.