Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

PREDICTING DEATH IN GAMES WITH DEEP REINFORCEMENT LEARNING

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
FRANCESCO BENINI

Sessione
Anno Accademico 2018-2019

Introduzione

Il contesto in cui si pone l'elaborato è una branca del machine learning, chiamato reinforcement learning [1]. Il reinforcement learning si compone di un agente che esegue delle azioni in un ambiente con l'obiettivo di massimizzare il reward, un valore numerico che si restituisce in seguito ad un passo d'esecuzione. Questo fine porta l'agente a seguire la migliore sequenza di azioni che gli garantisca la ricompensa maggiore. Inizialmente la conoscenza dell'ambiente è nulla, ma con l'esplorazione si acquisisce sempre più conoscenza, e questo produce una selezione delle strategie più vantaggiose.

Il reinforcement learning è un argomento che si è sviluppato molto recentemente e che ha visto anche la nascita di continue sfide ed altrettante soluzioni per affrontarle, una di queste e di fondamentale rilevanza è la combinazione con tecniche di deep learning, da cui nasce il Deep Reinforcement Learning (DRL). L'aggiunta delle reti neurali ha garantito un maggiore sviluppo dell'area. Un campo in cui si è sviluppato molto il DRL sono i giochi, visto che rientrano nella categoria di processi decisionali di Markov.

Quest'elaborato si pone come obiettivo di migliorare il lavoro sviluppato dal collega M. Conciatori [2], che utilizzava l'applicazione del reinforcement learning ai giochi Atari. In questa tesi ci si vuole soffermare sui giochi con ricompense molto sparse, dove la soluzione precedente non era riuscita a conseguire traguardi. I giochi con ricompense sparse sono quelli in cui l'agente prima di ottenere un premio, che gli faccia comprendere che sta eseguendo la sequenza di azioni corretta, deve compiere un gran numero di azioni. Tra i giochi con

queste caratteristiche, in questo elaborato ci si è focalizzati principalmente su uno, Montezuma's Revenge.

Montezuma's Revenge è stato al centro di molte sfide proprio a causa del suo elemento distintivo, cioè che per ottenere il primo reward è necessario eseguire un gran numero di azioni, e come se non bastasse c'è anche da considerare che il numero delle azioni tra cui scegliere ad ogni passo d'esecuzione è elevato, sono appunto 18. Per questo la totalità degli algoritmi sviluppati non è riuscita ad ottenere risultati soddisfacenti.

L'idea di proseguire il lavoro del collega M. Conciatori è nata proprio dal fatto che Lower Bound DQN [2] riusciva ad ottenere la prima ricompensa, senza, però, riuscire poi a ripetersi successivamente. Ci si è posti, perciò, come scopo principale di trovare una soluzione per poter ottenere risultati ottimali e si è, a tal fine, pensato di prevedere la morte dell'agente, aiutandolo, di conseguenza, ad evitare le azioni sbagliate e guadagnare maggiori ricompense. L'agente in questo contesto impiega più tempo per esplorare l'ambiente e conoscere quali comportamenti hanno un compenso positivo. In conseguenza di questo e vista la scarsità di ricompense positive, per il motivo precedentemente descritto, si è pensato di venire in aiuto dell'agente restituendogli una penalità per ciò che era dannoso al suo modo di agire, perciò, attribuendo una sanzione a tutte quelle azioni che causano la terminazione dell'episodio e quindi la sua morte. Le esperienze negative si memorizzano in un buffer apposito, chiamato *done buffer*, dal quale si estraggono poi per allenare la rete. Nel momento in cui l'agente si troverà nuovamente nella stessa situazione saprà quale azione sia meglio evitare, e con il tempo anche quale scegliere.

La modifica svolta a Lower Bound DQN ha restituito dei risultati non del tutto positivi, visto che per alcuni giochi il punteggio è rimasto invariato o comunque ha restituito valori simili a prima. Rispetto al caso specifico di Montezuma's Revenge, l'algoritmo è riuscito ad ottenere delle ricompense in numero maggiore rispetto a prima, ma non ha comunque convinto, perché ci

si aspettava risultati migliori.

Si descrive ora meglio nel dettaglio il contenuto dei singoli capitoli.
Nel **capitolo 1** si introduce tutto ciò che è il Background dell'argomento della tesi. Si inizia con una breve descrizione dell'intelligenza artificiale e del machine learning. Si prosegue poi con una descrizione più nel dettaglio di cos'è il reinforcement learning e come funziona. Infine si descrivono gli ambienti in cui si è eseguito l'algoritmo.
Nel **capitolo 2** si espone l'algoritmo del Q-learning [6], che è la base rispetto agli algoritmi value-oriented del reinforcement learning. Si descrive poi DQN (Deep Q-Network)[7], uno dei primi algoritmi nati con l'aggiunta delle reti neurali e da cui ha inizio anche l'algoritmo del collega. Si passa, perciò, a descrivere l'origine del progetto sviluppato dal collega M. Conciatori, per poi proseguire con le modifiche da lui eseguite, entrando nel dettaglio rispetto agli hyperparameters ed ai valori usati per eseguire il modello.
Nel **capitolo 3** si propongono tutte le variazioni attuate ai fini di questo elaborato rispetto all'algoritmo iniziale Lower Bound DQN. Il tutto focalizzandosi sull'obiettivo che è predire la morte degli agenti nei giochi Atari e discutendo i cambiamenti applicati agli hyperparameters. Nella parte conclusiva del capitolo si pone, infine, risalto sulle tecnologie utilizzate.
Nel **capitolo 4** si presentano tutti i risultati ottenuti confrontati direttamente con quelli dell'implementazione iniziale di Lower Bound DQN.
Nel **capitolo 5** si descrivono le difficoltà riscontrate durante l'implementazione del progetto e la sua sperimentazione. Si conclude suggerendo possibili miglioramenti per sviluppi futuri.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Background

1.1 Introduzione al Machine learning

Il Machine Learning è un ramo dell'Intelligenza Artificiale che studia algoritmi in grado di apprendere senza essere stati esplicitamente e preventivamente programmati, tramite inferenza e con l'utilizzo di dati di allenamento. Il termine nacque da Arthur Samuel, pioniere dell'intelligenza artificiale, che nel 1959 ne diede una prima definizione [8]:

> *"campo di studio che dà ai computer l'abilità di apprendere (di realizzare un compito) senza essere esplicitamente programmati a farlo".*

Nell'approccio di sviluppo tradizionale si scrive una funzione per produrre un output avendo noti l'input e l'algoritmo.

Nel machine learning invece si conosce l'input e l'output desiderato ma non si conosce l'algoritmo che porta a quel risultato.

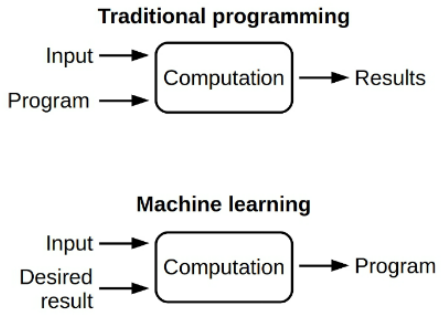

Figura 1.1: Programmazione tradizionale vs Machine Learning

1.1.1 Come avviene l'apprendimento automatico?

Allo stesso modo in cui gli umani apprendono dalle esperienze passate, lo stesso vale per il machine learning, vengono forniti molti esempi (dati) e la macchina inizia a capire cosa sta succedendo. Nel machine learning il comportamento degli algoritmi non è pre-programmato ma appreso dai dati. Il processo di apprendimento avviene in vari modi:

- **dall'osservazione dell'ambiente esterno**
  l'agente osserva il mondo esterno e impara dai feedback delle sue azioni e dai suoi errori.

- **dalla base di conoscenza**
  l'esperienza e la conoscenza dell'ambiente operativo sono conservate in un database detto base di conoscenza (knowledge base o KB).

Queste due modalità di apprendimento sono complementari.

1.1.2 Le tecniche del machine learning

Le principali tecniche di machine learning sono le seguenti:

1. **Apprendimento supervisionato (supervised learning)**
   Vengono forniti alla macchina una serie di esempi. Ognuno di questi è composto da una sequenza di valori di input ed è accompagnato da un'etichetta in cui viene indicato il risultato. Questa elabora i dati e apprende dagli esempi per individuare una funzione predittiva o una regola generale.

   (a) Perché si chiama supervisionato?
   Per *supervisione* si intende la presenza delle soluzioni (etichette) nell'insieme dei dati di addestramento. Una persona (supervisore) fornisce alla rete degli esempi pratici, ciascuno dei quali ha indicate le variabili di input (x) e il risultato corretto (y). La rete impara dai prototipi forniti ed elabora un modello predittivo. Nel-

l'apprendimento supervisionato l'obiettivo è stimare una funzione $f(x)$ incognita che collega le variabili di input x a una variabile di output y.

$$y = f(x) \tag{1.1}$$

Non si conosce la funzione $f(x)$ a priori e si pone come scopo di stimare una funzione ipotesi $h(x)$ in grado di approssimare la $f(x)$.

$$y = h(x) \tag{1.2}$$

Per farlo si analizza un insieme di dati detto *training set*[1]fornito dal supervisore. Un training set è un insieme di addestramento composto da un insieme di coppie (x, y).

(b) Come capire se la funzione ipotesi è corretta?
Si deve valutare l'accuratezza della funzione ipotesi $h(x)$, se approssima o meno la funzione $f(x)$ (che non si conosce).
Per capirlo si utilizza un altro insieme di dati, detto *test set*[1], fornito sempre dal supervisore. A questo punto la rete risponde agli n esempi del test set. Poi confronta ogni sua risposta con la risposta corretta indicata dagli esperti (Y) nel test set. Se la percentuale di risposte corrette $\frac{Risposte\,corrette}{Numero\,test}$ della rete è soddisfacente, allora la funzione ipotesi $h(x)$ supera l'esame e viene accolta. In caso contrario l'apprendimento supervisionato riparte con l'analisi di un ulteriore training set e il ciclo ricomincia da capo.

2. **Apprendimento non supervisionato (unsupervised learning)**
Nell'apprendimento non supervisionato l'agente non ha esempi di input-output da testare, vengono forniti solo i dati di input. Non c'è un istruttore a decidere se un'azione è corretta o no, né una misura di riferimento per i feedback dell'ambiente. L'algoritmo cerca pattern e similitudini fra i dati forniti, scegliendo in autonomia i criteri di raggruppamento.

[1]Sia il *training set* che il *test set* sono forniti dal supervisore umano.
Tuttavia, i due insiemi sono composti da esempi diversi.

1.2 Introduzione al Reinforcement learning

Il *reinforcement learning* è imparare cosa fare - come mappare situazioni e azioni - al fine di massimizzare il valore numerico di ricompensa.
L'agente non ha esempi di input e output, né una supervisione che gli spieghi il comportamento corretto da seguire. Tuttavia deve scoprire quali azioni producono il maggior *reward*, provandole [1].
In questo modo, con l'esperienza l'agente costruisce una funzione di comportamento in grado di massimizzare il reward.

Nei casi più interessanti, le azioni possono influenzare non solo il reward immediato ma anche situazioni successive e, attraverso ciò, tutte le ricompense successive. Queste due - *trial-and-error search and delayed reward* - sono le caratteristiche più importanti, distintive del reinforcement learning.

Il reinforcement learning è di solito formalizzato usando idee dalla teoria dei sistemi dinamici, nello specifico il processo decisionale di Markov (Finite Markov Decision Process). Un agente deve essere in grado di percepire lo stato del suo ambiente in una certa misura e intraprendere azioni che influenzano quello stato. L'agente deve anche avere uno o più obiettivi relativi allo stato dell'ambiente.
I processi decisionali di Markov includono questi tre aspetti - *sensazione, azione e obiettivo* - nelle loro forme più semplici possibili. Qualsiasi metodo che sia adatto a risolvere tali problemi viene considerato un metodo di reinforcement learning.

Un altro aspetto che caratterizza il RL è il compromesso tra *exploration and exploitation*: la scelta tra tentare azioni con risultati sconosciuti per vedere come influenzano l'ambiente oppure eseguire azioni con un esito atteso in base alle conoscenze già disponibili. Per ottenere le migliori prestazioni, l'agente deve sfruttare ciò che ha già sperimentato per ottenere un reward (*exploit*), ma deve anche esplorare per effettuare selezioni di azioni migliori in futuro (*explore*). Il dilemma è che né l'*exploration* e né l'*exploitation* possono essere

eseguite in modo esclusivo senza fallire nel compito. L'agente deve provare una varietà di azioni e favorire progressivamente quelle che sembrano essere le migliori.

Il RL è diverso dal supervised learning, in quanto non si basa su un *training set* fornito da un supervisore esterno. È anche diverso dal unsupervised learning, poiché il suo obiettivo è massimizzare il valore numerico di reward.

Un'altra caratteristica interessante del reinforcement learning è la sua interazione con molte discipline ingegneristiche e scientifiche che appartengono ai campi di intelligenza artificiale, statistica, ottimizzazione, ricerca operativa, teoria del controllo e altre materie matematiche, ma anche psicologia, biologia e neuroscienza.
Tra tutte le forme di machine learning, il reinforcement learning è quello più vicino al tipo di apprendimento di umani e animali. Molti degli algoritmi fondamentali sono ispirati ai sistemi di apprendimento biologico.

1.2.1 Finite Markov Decision Process (FMDP)

In questa sezione verrà descritto il processo decisionale di Markov (FMDP). Gli MDP sono una formalizzazione classica del processo decisionale sequenziale, dove le azioni influenzano non solo i reward immediati, ma anche le situazioni o stati successivi, e attraverso questi i reward futuri.

1.2.2 L'interfaccia agente-ambiente

Gli MDP sono pensati per dare una chiara definizione del problema dell'apprendimento dall'interazione per poter raggiungere un obiettivo. Per poter descrivere gli MDP è necessario descrivere innanzitutto lo schema di interazione tra chi apprende, chiamato *agent* e l'ambiente esterno con cui l'agente interagisce denominato *environment*. Questi interagiscono continuamente: l'agente selezionando le azioni e l'ambiente rispondendo a queste azioni e

presentando nuove situazioni all'agente. L'ambiente crea i reward, valori numerici che l'agente cerca di massimizzare nel tempo attraverso la scelta delle azioni.

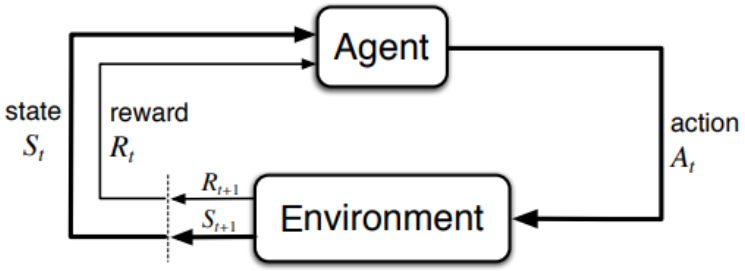

Figura 1.2: Agente che interagisce con l'ambiente nell'ambito del RL

Nello specifico, l'agente e l'ambiente interagiscono in ciascuna sequenza di passi temporali discreti, $t = 0, 1, 2, 3...$ dove ad ogni passo t, l'agente riceve una rappresentazione dello *stato* dell'ambiente, $S_t \in \mathcal{S}$ e basandosi su quella sceglie un' *azione* $A_t \in \mathcal{A}(s)$ dal set di tutte le azioni possibili. Nello step successivo, come conseguenza della sua azione, l'agente riceve un reward numerico, $R_{t+1} \in \mathcal{R}$ e si ritrova in uno stato nuovo, S_{t+1}.

In un FMDP, i set degli stati, azioni e rewards ($\mathcal{S}, \mathcal{A}$, e $\mathcal{R}$) hanno un numero finito di elementi. Inoltre le variabili random R_t e S_t hanno distribuzioni di probabilità discrete che dipendono solo dal precedente stato e azione, seguendo la proprietà Markov.

Definizione 1 Un processo discreto di controllo stocastico a tempo ha le proprietà di Markov se

$$P(S_{t+1}) \mid S_t = P(S_{t+1}) \mid S_1, ..., S_t \tag{1.3}$$

Un processo decisionale Markov è un processo di controllo stocastico a tempo discreto definito come segue:

Definizione 2 Un FMDP è una quintupla $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, dove:

- $\mathcal{S}$ è un set di stati *finito*
- $\mathcal{A}$ è un set di azioni *finito*
- $\mathcal{P}$ è una *funzione di probabilità della transizione dello stato*
  $P : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$
  $p(s' \mid s, a) = P(S_t = s' \mid S_{t-1} = s, A_{t-1} = a)$ esprime la probabilità che l'azione a nello stato s al tempo $t - 1$ conduca allo stato s' al tempo t.
- $\mathcal{R}$ è la *funzione di reward*: $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
  $r(s, a) = \mathbb{E}[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r \mid s, a)$
- γ è un *fattore di discount*, $\gamma \in [0, 1]$

1.2.3 Obiettivi e Reward

Nel reinforcement learning lo scopo o l'obiettivo (*goal*) dell'agente è formalizzato in termini di un segnale speciale, chiamato *reward*, che passa dall'ambiente all'agente. In ogni passo, il reward è un semplice valore numerico, $R_t \in \mathbb{R}$.

In modo informale, l'obiettivo dell'agente è quello di massimizzare l'importo totale del reward che riceve. Ciò significa massimizzare la ricompensa non immediata, ma una ricompensa cumulativa a lungo termine. Si può affermare chiaramente questa idea come *ipotesi di reward*:

> Tutto ciò che intendiamo per *goals* può essere pensato come la massimizzazione del valore atteso della somma cumulativa di un segnale scalare ricevuto (chiamato *reward*) [1].

Il modo in cui si forma il reward influenza direttamente il comportamento dell'agente, ecco perché la definizione del *reward* è un punto chiave quando si definisce un modello di reinforcement learning.

1.2.4 Valori di ritorno ed episodi

Per esprimere in modo più formale l'idea del reward cumulativo è necessario introdurre il concetto del *valore di ritorno atteso* che si cerca di massimizzare, indicato G_t. Nei casi più semplici il valore di ritorno è la somma dei reward dove T è l'ultimo step temporale.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{1.4}$$

Questo approccio ha senso quando l'interazione agente-ambiente si scompone in sotto sequenze, che vengono chiamate *episodi*, come le partite di un gioco, i viaggi attraverso un labirinto o qualsiasi tipo di interazione ripetuta.
Ogni episodio termina in uno stato speciale chiamato *stato terminale*, seguito da un reset a uno stato di partenza standard. Anche se si pensa che gli episodi finiscano in modi diversi, come vincere e perdere una partita, l'episodio successivo inizierà indipendentemente da come è finito quello precedente. In questo modo gli episodi possono finire nello stesso stato terminale, con reward diversi per i diversi risultati.
I task con questo tipo di episodi sono chiamati *episodic task*. In questi task è necessario distinguere l'insieme di tutti gli stati non terminali, indicato con $\mathcal{S}$, dall'insieme di tutti gli stati più lo stato terminale, indicato con $\mathcal{S}^+$. Il tempo, T, è una variabile random che normalmente varia da episodio a episodio.

D'altra parte, in molti casi l'interazione agente-ambiente non si distingue in episodi identificabili, ma va avanti continuamente senza limiti. Ad esempio, questo sarebbe il modo più naturale per definire un'attività di controllo del processo in corso, o un'applicazione a un robot con una lunga durata di vita. Questi task sono chiamati *task continui*.

La formula 1.4 citata precedentemente è problematica per i task continui in quanto il passo finale del tempo sarebbe $T = \infty$ e il valore di ritorno, che è ciò che si sta cercando di massimizzare, potrebbe portare a un reward infinito.

Per questo motivo, la formula del valore di ritorno si avvale della definizione di *discount*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{1.5}$$

dove γ è il parametro, compreso nell'intervallo $0 \leq \gamma \leq 1$, chiamato *discount rate*.

Il discount rate stima il valore attuale dei reward futuri, ovvero, funziona come un "peso" sulla loro importanza, dove un reward ricevuto k step nel futuro vale solo $k+1$ volte quello che varrebbe se fosse ricevuto immediatamente.

- Se $\gamma < 1$, la sommatoria del (1.5) ha un valore finito finché il reward R_k è delimitato.
- Se $\gamma = 0$, l'agente è "miope" nel guardare solo la massimizzazione del reward immediato: l'obiettivo è quello di capire come scegliere A_t per rendere massimo R_{t+1}. Se ognuna delle azioni dell'agente è avvenuta per influenzare solo il reward immediato, e non anche quello futuro, allora un agente "miope" potrebbe massimizzare (1.5), rendendo massimo separatamente ogni reward immediato. Ma, in generale, agire per ottenere il reward immediato più alto possibile può ridurre l'accesso ai reward futuri facendo sì che il valore di ritorno sia ridotto.
- Con l'approssimarsi di γ a 1, l'agente tiene conto del reward futuro diventando così più lungimirante.

1.2.5 Funzione di valore e policies

Quasi tutti gli algoritmi di reinforcement learning implicano la stima di *funzioni di valore* -funzioni di stati (o di coppie stato-azione) che stimano quanto sia buono per l'agente essere in un determinato stato (o quanta bontà abbia eseguire una data azione in un dato stato). Questa funzione di valore vie-

ne valutata a fronte di una *policy*, che informalmente, è il comportamento dell'agente.

Formalmente, una *policy* è mappare dagli stati alle probabilità di selezionare ogni possibile azione. Se l'agente sta seguendo la policy π al tempo t, allora $\pi(a \mid s)$ è la probabilità che $A_t = a$ se $S_t = s$.

$$\pi(a \mid s) = p(A_t = a \mid S_t = s) \tag{1.6}$$

Come p, π è una funzione ordinaria; "|" in mezzo a $\pi(a \mid s)$ ricorda che definisce una distribuzione di probabilità su $a \in \mathcal{A}(s)$ per ogni $s \in \mathcal{S}$. I metodi di RL specificano come la policy dell'agente viene cambiata in seguito alla sua esperienza.

La *funzione di valore* di uno stato s con la policy π, denotato $v_\pi(s)$, è il valore di ritorno previsto iniziando in s e seguendo π in seguito [1]. Si definisce v_π come segue:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s\right] \text{ per ogni } s \in \mathcal{S} \tag{1.7}$$

dove $\mathbb{E}_\pi[.]$ indica il valore atteso di una variabile casuale dato che l'agente segue la policy π, e t è lo step nel tempo. La funzione v_π è chiamata *funzione stato-valore* per la *policy* π.

Si può definire, in modo simile, il valore di intraprendere un'azione a in stato s con la policy π, definito $q_\pi(s, a)$ ovvero il valore di ritorno previsto a partire da s, intraprendendo l'azione a e successivamente seguendo la policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a\right] \tag{1.8}$$

In questa formula, q_π è la *funzione stato-valore* per la *policy* π.

Infine, è possibile definire una terza funzione di valore, chiamata *funzione di vantaggio* che utilizza contemporamente la *funzione stato-valore* q e la funzione di valore v per descrivere quanto è buona l'azione a in confronto al valore di ritorno atteso, seguendo la policy π.

1.2.6 Equazione di Bellman

Una proprietà fondamentale delle funzioni di valore utilizzate durante il reinforcement learning sta nel fatto che esse soddisfano relazioni ricorsive simili a quelle che si è descritto precedentemente. Per ogni policy π e ogni stato s, la seguente condizione esprime il legame tra il valore di s e il valore dello stato successivo:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t \mid S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} \mid S_{t+1} = s'] \Big] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big], \text{ per ogni } s \in \mathcal{S} \end{aligned} \tag{1.9}$$

dove:

- l'azione a viene presa dal set $\mathcal{A}(s)$
- gli stati successivi s' vengono presi dal set $\mathcal{S}$
- i reward vengono presi dal set $\mathcal{R}$

L'equazione appena descritta è *l'equazione di Bellman per* v_π che descrive la relazione tra il valore di uno stato ed i valori degli stati successori. Il principio è quello di guardare avanti da uno stato ai suoi possibili stati successori, come suggerito dalla seguente figura.

Ogni cerchio bianco rappresenta uno stato e ogni cerchio nero rappresenta una coppia stato-azione. Partendo dallo stato s, la radice, l'agente potrebbe intraprendere una qualsiasi di queste azioni- tre sono quelle mostrate nel

diagramma- in base alla sua policy π. Da ognuno di questi, l'ambiente potrebbe rispondere con uno qualsiasi dei diversi stati successivi, s', insieme a un reward, r, secondo la distribuzione p.

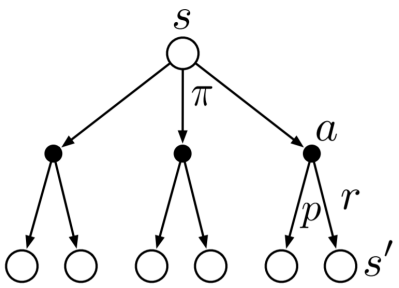

Figura 1.3: Diagramma di backup per v_π

L'equazione di Bellman (1.9) fa una media di tutte le possibilità, ponderando ciascuna di esse in base alla sua probabilità di verificarsi. Afferma che il valore dello stato di partenza deve essere uguale al valore (ridotto) del prossimo stato atteso, a cui va aggiunto il reward atteso lungo il percorso. La funzione valore v_π è l'unica soluzione all'equazione di Bellman. Quest'equazione rappresenta la base di una serie di modi per calcolare, approssimare e studiare v_π. Il diagramma mostrato in figura 1.3 viene chiamato *diagramma di backup* in quanto schematizza le relazioni che costituiscono la base delle operazioni di aggiornamento o di backup che sono il fulcro del metodo di reinforcement learning [1].

1.2.7 Ottimalità

Risolvere un task di reinforcement learning significa trovare una policy che raggiunga reward alti a lungo termine. Per gli FMDP, si può definire con precisione una policy ottimale. Le funzioni di valore definiscono un ordine parziale sulle policy.

Una policy π è definita essere migliore o uguale a una policy π' se il suo valore di ritorno atteso è superiore o uguale a quella di π' per tutti gli stati. In altre parole, $\pi \geq \pi'$ se e solo se $v_\pi(s) \geq v_{\pi'}(s)$ per ogni $s \in \mathcal{S}$.
C'è sempre almeno una policy migliore o uguale a tutte le altre policy. Questa è una *policy ottimale*, indicata da π_*.

Tutte le policy ottimali condividono la stessa funzione ottimale stato-valore

v_*, definito come segue:

$$v_*(s) = \max_\pi v_\pi(s), \text{ per ogni } s \in \mathcal{S} \tag{1.10}$$

Le policy ottimali, condividono anche la stessa funzione ottimale azione-valore, indicata da q_* e definito come segue:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \text{ per ogni } s \in \mathcal{S}; a \in \mathcal{A} \tag{1.11}$$

Per la coppia stato-azione (s,a), la funzione restituisce il valore atteso per l'azione a nello stato s seguendo una policy ottimale. In questo modo, si può scrivere q_* in termini di v_* come segue:

$$q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{1.12}$$

Poiché v_* è la funzione di valore per una policy, deve soddisfare la condizione di auto consistenza data dall'equazione di Bellman (1.9) per i valori di stato. Dato che è una funzione di valore ottimale, la condizione di consistenza di v_* può essere scritta in una forma speciale senza riferimento ad alcuna policy. Di seguito, *l'equazione Bellman ottimale* per v_* :

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi*}(s,a) \\ &= \max_a \mathbb{E}_{\pi*}[G_t \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi*}[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a] \\ &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s')] \end{aligned} \tag{1.13}$$

L'equazione Bellman ottimale per q_* è la seguente:

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned} \tag{1.14}$$

I diagrammi di backup in figura mostrano graficamente gli archi dei futuri stati e le azioni considerate nell'equazione di Bellman ottimale per v_* e q_*. Sono stati aggiunti ai diagrammi (che sono simili a quelli in figura 1.3) gli archi nei nodi di scelta dell'agente per rappresentare che è stato scelto il massimo rispetto al valore previsto data una policy.

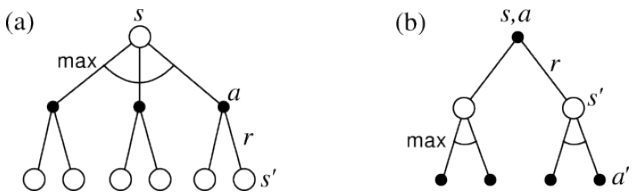

Figura 1.4: Diagramma di backup per v_* e per q_*

Il diagramma di backup a sinistra rappresenta l'equazione di Bellman (1.13) e quello a destra rappresenta l'equazione (1.14).

1.3 Atari 2600

Atari 2600 è una console di gioco creata nel 1977, per la quale negli anni sono stati creati diversi giochi. Questo tipo di giochi ha caratteristiche rilevanti su cui applicare algoritmi di reinforcement learning. Per questo, sin dalla nascita dei primi algoritmi sono stati utilizzati come ambienti di prova, ed ora possono essere considerati uno standard in questo campo di ricerca. Inoltre, la creazione di framework che semplificassero il loro utilizzo ne ha continuamente aumentato l'uso, facendo sì che per valutare un nuovo algoritmo si confrontassero i risultati precedentemente ottenuti.

1.3.1 Arcade Learning Environment (ALE)

La Arcade Learning Environment [4] è stata pubblicata nel 2013 e serviva per permettere ai ricercatori, e a chi ne volesse far uso, di sviluppare agenti intelligenti.

La ALE si basa sui giochi Atari 2600. L'emulatore è stato creato con lo scopo di offrire a chi volesse utilizzare modelli di intelligenza artificiale dei giochi, con degli ambienti già pronti all'utilizzo e con caratteristiche basilari, separando i dettagli dell'ambiente dalla progettazione dell'agente.

Prima dell'introduzione di queste risorse, per l'utilizzo di algoritmi di reinforcement learning, era necessario costruirsi autonomamente tutte le caratteristiche da dare in input ad un agente, le quali inoltre valevano solo per il singolo gioco.

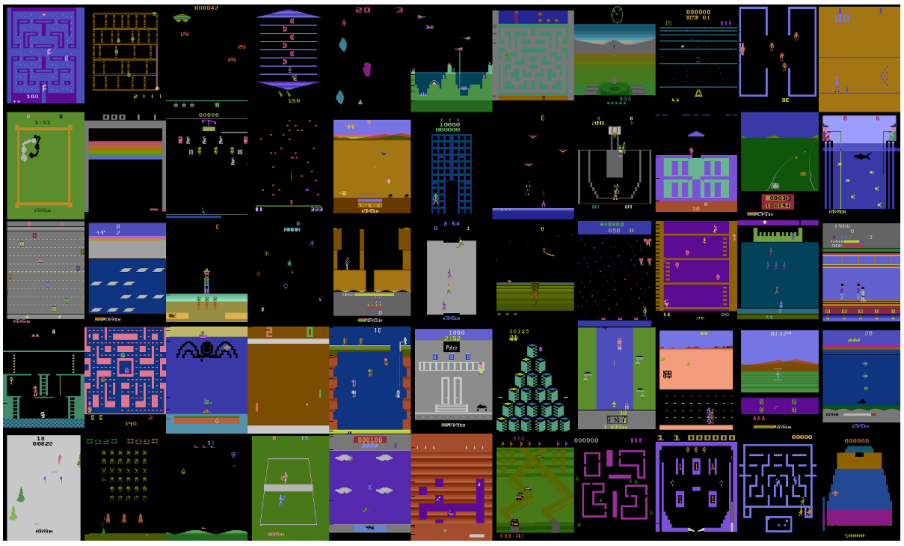

Figura 1.5: Giochi ATARI disponibili in ALE

1.3.2 OpenAI Gym

Gym [5] è un toolkit creato da OpenAI, il quale viene in aiuto agli sviluppatori che lo possono usare per confrontare i diversi algoritmi di reinforcement learning. Questo strumento supporta l'insegnamento agli agenti di qualunque cosa, dal camminare fino a giocare giochi complessi. É un'interfaccia open source per eseguire i compiti del reinforcement learning.

All'interno di Gym ci sono diversi environment, in particolare i giochi di Atari 2600, i quali possono essere utilizzati dallo sviluppatore per implementare e testare i propri algoritmi, come è stato fatto in questo progetto. Colui che implementa può usare una qualsiasi libreria di computazione numerica, come per esempio Tensorflow. L'utilizzo è semplice e intuitivo. All'inizio si crea un environment scegliendo tra quelli esistenti. Gym contiene tutte le informazioni relative ad ogni singolo ambiente: il numero di azioni di cui l'agente dispone, il numero di vite a disposizione, ecc. Successivamente si esegue la funzione *step(azione)*, alla quale viene passata l'azione scelta, e vengono restituiti: la nuova osservazione, il reward ottenuto, una variabile *done*, che indica se l'azione ha portato alla conclusione dell'episodio, ed un dizionario al cui interno vengono inserite delle informazioni di diagnostica.

Quando la variabile *done* sarà *true* e quindi l'episodio terminato, avverrà il *reset* dell'ambiente per procedere con il nuovo episodio e le nuove esperienze. Ci sono due modalità per valutare i risultati ottenuti, questo perché si è scelto di standardizzare per poter confrontare le conclusioni prodotte:

- `No-op starts`: L'agente seleziona l'azione "fai niente" fino a 30 volte all'inizio dell'episodio. Questo permette di contribuire alla causalità della posizione iniziale dell'agente e rendere ogni esecuzione leggermente diversa una dalle altre.

- `Human starts`: L'agente inizia il gioco da uno dei 100 punti di partenza raggiunti da un giocatore umano professionista. Questo consente di rendere ogni episodio molto differente l'uno dall'altro.

Ogni gioco Atari è disponibile in più versioni: *v0* vs *v4*, *Deterministic* vs *NoFrameskip* e *ram*:

- `v0` indica che sono usate le *sticky actions*, le quali rendono imprevedibile l'azione scelta dall'agente, mentre `v4` indica l'utilizzo di azioni normali.
- `Deterministic` denota l'applicazione di un frameskip di 4 frame, al contrario `NoFrameskip`, come dice il nome, rappresenta l'assenza di frameskip. La mancanza di entrambi comporta un frameskip casuale di 2, 3 o 4.
- Con `Ram`, infine, l'input fornito all'algoritmo è composto dal contenuto delle celle di ram che riguardano lo stato del gioco.

1.3.3 Montezuma's Revenge

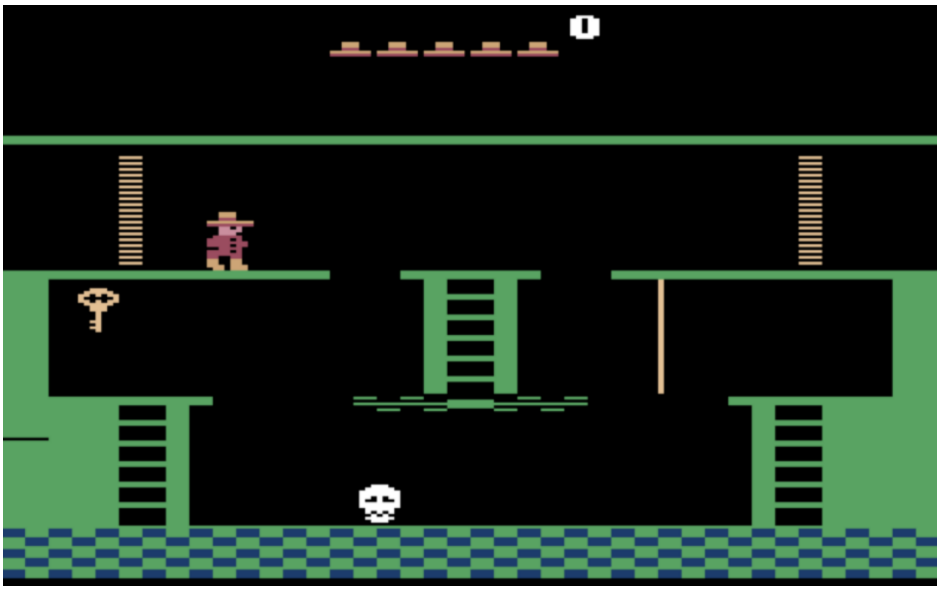

Figura 1.6: Montezuma's Revenge.

Montezuma's Revenge è uno dei giochi che appartiene ad Atari 2600 e che quindi sono presenti in OpenAI Gym.

Questo gioco si distingue dagli altri per la sua caratteristica di essere un gioco a reward molto sparso, questo significa che l'agente riceve il reward unicamente dopo aver completato una serie specifica di azioni in un certo arco di tempo.

In aggiunta, a complicare questa situazione, le possibilità di fallimento all'interno del gioco sono molteplici, sono infatti diversi i modi in cui l'agente può morire, ed il più semplice di questi è quello di cadere da troppo in alto.

A causa di queste difficoltà, Montezuma's Revenge ha rappresentato diverse challenge nell'ambito del Deep Reinforcement Learning, ispirando lo sviluppo di molti tra i più interessanti approcci per estendere e rimodulare il tradizionale Deep RL, in particolare nell'ambito di *hierarchical control*, *exploration* ed *experience replay*.

Questo gioco, vista la sua particolarità e le diverse sfide che propone, è stato quello su cui ci si è focalizzati maggiormente per provare ad ottenere risultati soddisfacenti.

Capitolo 2

DQN

La tesi prende inizio dall'elaborato del collega M. Conciatori, il cui lavoro si basa sull'algoritmo DQN [7]. Aggiungendo le tecniche di deep learning a Q-learning [6] nasce l'algoritmo DQN (Deep Q-Network), grazie al quale è stato possibile affrontare problemi più complessi. In questo capitolo si descrivono gli algoritmi Q-learning e DQN, introdotti prima. Si prosegue con l'esposizione di OpenAI baselines [17], che è il framework da cui è stata acquisita la versione iniziale di M. Conciatori. Ed infine ci si sofferma su Lower Bound DQN, una delle versioni implementate dal collega e l'algoritmo da cui si è partiti per lo sviluppo di questo progetto.

2.1 Q-learning

Q-learning [6] è un algoritmo di reinforcement learning off-policy che cerca di trovare l'azione migliore da eseguire dato lo stato corrente. É considerato off-policy perché la funzione di Q-learning impara da azioni che non fanno parte dell'attuale policy, come per esempio l'esecuzione di azioni random.
Nello specifico, Q-learning tenta di imparare una policy che massimizzi il reward totale.

La 'Q' in Q-learning rappresenta la qualità, cioè quanto una certa azione è utile per far ottenere all'agente futuri reward. Quando l'algoritmo viene eseguito si crea una q-table, rappresentata sotto forma di matrice che mappa ogni coppia (stato, azione) in un valore numerico, inizialmente settato a zero. Dopo ogni episodio i Q-values vengono aggiornati e memorizzati.
La q-table diventa una tabella di riferimento per l'agente, il quale in base allo stato seleziona l'azione migliore da eseguire.

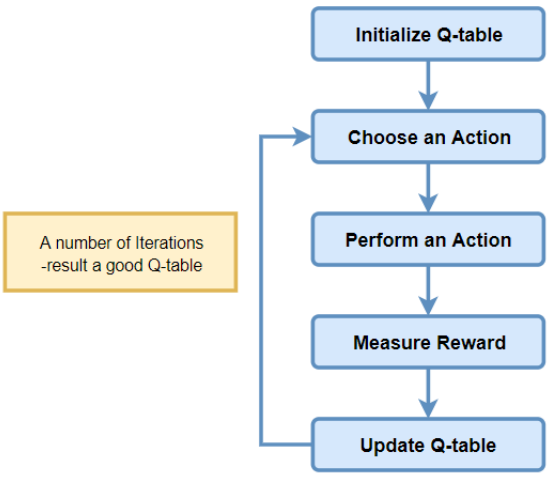

Figura 2.1: Processo Q-learning.

L'interazione dell'agente con l'ambiente contribuisce ad aggiornare i valori all'interno della q-table. L'agente può interagire con l'ambiente in due modalità, che sono basate sul compromesso tra *exploiting* e *exploring*, descritto nel *capitolo 1*:

- La prima si basa sull'utilizzo della q-table da cui l'agente va a selezionare l'azione che per quello stato ha il valore maggiore, cioè l'`exploiting`, visto che si usa la conoscenza a nostra disposizione per prendere una decisione.
- L'altro modo per scegliere l'azione da eseguire è farlo randomicamente, cioè l'`exploring`. Invece che scegliere un'azione che possa massimizzare il reward futuro, si seleziona l'azione casualmente. L'esplorazione è

importante perché permette all'agente di conoscere meglio l'ambiente e scoprire nuovi stati che altrimenti non potrebbero essere raggiunti se si scegliesse sempre l'azione con il valore più alto.

Come già menzionato, è importante trovare un bilanciamento tra questi due procedimenti, per farlo si usa una variabile ε, la quale si setta in base alla proporzione che si vuole ottenere.

Gli aggiornamenti avvengono dopo ogni passo o azione e finiscono con la conclusione dell'episodio, per poi ricominciare con quello successivo. L'episodio si conclude raggiungendo un punto terminale, che per un gioco è la fine della partita, quindi la vittoria o la sconfitta, che può coincidere anche con la morte dell'agente. Dopo un singolo episodio l'agente non avrà imparato molto, in quanto necessita di diversi episodi per esplorare ed apprendere, per poi convergere su valori ottimali di Q-values.

Ci sono un paio di nozioni aggiuntive da menzionare per descrivere come avviene l'aggiornamento dei Q-values. In particolare, il nuovo valore dei Q-values non sarà una semplice sostituzione del vecchio, ma si basa sulla differenza tra il nuovo valore ridotto ed il vecchio valore:

$$Q(s,a) = Q(s,a) + \alpha * (r' + \gamma * \max_a Q(s',a') - Q(s,a)), \qquad (2.1)$$

dove *r'*, *s'* e *a'* rappresentano rispettivamente il reward, l'osservazione e l'azione al passo successivo.

Il nuovo valore viene ridotto usando una variabile gamma (γ), la stessa descritta nel *capitolo 1*, ed inoltre viene regolata la dimensione del passo usando il learning rate (lr).

Gamma: chiamato discount factor, è usato per bilanciare il peso tra il reward attuale e quello futuro. Si applica direttamente al nuovo reward ottenuto ed ha solitamente un valore che si aggira tra 0.8 e 0.99. Più il valore sarà alto, più verrà data importanza al reward futuro.

Learning Rate: spesso rappresentato tramite α, si può definire semplicemen-

te come una misura che rappresenta quanto si voglia tenere in considerazione il nuovo valore rispetto al vecchio. Come si può vedere sopra, il valore si moltiplica per il risultato della differenza tra il valore nuovo e vecchio. Infine si applica il valore ottenuto al Q-value precedente, che quindi si muove nella direzione dell'ultimo aggiornamento.

2.2 Deep Q-Network (DQN)

Nonostante Q-learning sia un algoritmo molto potente, il suo principale punto debole è la carenza di generalizzazione. Questo avviene perché l'algoritmo di Q-learning si occupa unicamente di aggiornare i valori all'interno della tabella, perciò per quegli stati che l'agente non ha mai incontrato, non ha idea di che azione eseguire. Di conseguenza, non ha l'abilità di stimare il valore per stati mai visti.

Per ovviare a questo problema, DQN [7] sostituisce l'array a 2-dimensioni usato per memorizzare i valori, con una rete neurale. DQN si appoggia alla rete neurale per stimare i Q-values. Viene dato in input alla rete il frame della situazione corrente del gioco, mentre l'output è rappresentato dai Q-values delle azioni.

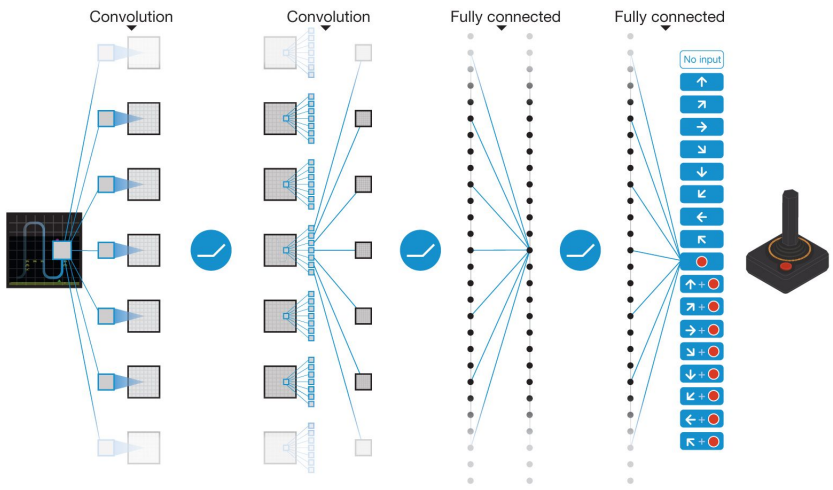

Figura 2.2: Architettura DQN.

Essendo l'input il frame del gioco, questo permette di rendere la struttura della rete indipendente dal problema in esame.
L'unica differenza è nell'output, che si differenzia in base al numero delle azioni disponibili nell'ambiente. La rete neurale che l'input attraversa è formata da strati con reti convoluzionali e livelli completamente connessi.
Il training della rete avviene basandosi sull'equazione di update del Q-learning. La funzione di loss è definita come l'errore quadratico tra i target Q-values ed i Q-values:

$$(r' + \gamma * \max_a \bar{Q}(s', a') - Q(s, a))^2, \tag{2.2}$$

dove $\bar{Q}$ rappresenta la target network.

Ci sono anche altre due tecniche che sono alla base per il training di DQN:

- `Experience Replay`: Visto che gli esempi di training in un tipico modello RL sono altamente correlati, questo renderà difficile la convergenza della rete. Per risolvere questo problema di distribuzione degli esempi si adotta experience replay. Essenzialmente, i singoli passi dell'agente vengono memorizzati, e se ne estrae casualmente una parte per aggiornare la conoscenza.
- `Target Network`: La target Q Network ha la stessa struttura di quella che stima i Q-values. Ogni volta che viene realizzata una certa quantità di passi, la target network viene resettata con i pesi della rete principale. Pertanto, le fluttuazioni diventano meno rilevanti, dando risultato ad un training più stabile.

Per usare *experience replay* i dati vengono registrati in un buffer, dove sono salvati come tuple nel formato $(s_t, a_t, r_t, s_{t+1}, done)$, dove:

- s_t è lo stato, cioè l'osservazione dell'ambiente, al tempo t;
- a_t è l'azione svolta nello stato s al tempo t;
- r_t è il reward ricevuto dopo aver eseguito l'azione a_t;
- s_{t+1} è lo stato al tempo t+1, restituito dopo aver eseguito a_t;

- *done* è una variabile booleana che ci dice se l'azione ha portato alla conclusione dell'episodio, se quindi s_t è uno stato terminale.

Esistono poi degli ulteriori procedimenti da considerare. Il *clipping rewards* viene eseguito per uniformare i risultati tra tutti i giochi, mentre il *preprocessing delle immagini* ed il *frame skipping* per adattare meglio l'input alla rete.

- `Clipping rewards`: Si impiega per standardizzare i punteggi dei giochi, visto che diversi giochi hanno differenti scale di valutazione. La procedura applica come soluzione quella di normalizzare tutti i rewards positivi su +1 e quelli negativi a -1. In assenza di rewards viene restituito il valore neutro 0.

- `Preprocessing delle immagini`: le immagini prima di essere consegnate alla rete hanno necessità di un'operazione di *preprocessing*. Questo si compone di più passi:

  1. Si fanno apparire in tutti i frame gli oggetti;
  2. Si rendono in scala di grigi;
  3. Si ridimensionano, trasformandoli in 84×84.

- `Frame skipping`: ALE è in grado di restituire 60 immagini al secondo. Però attualmente l'agente non esegue molte azioni in un secondo. In conclusione questa procedura permette a DQN di stimare i Q-values ogni 4 frame, inoltre garantisce di ridurre i costi computazionali e di raccogliere più esperienze.

2.3 OpenAI Baselines

In rete esistono diversi progetti che implementano molti algoritmi usati nel reinforcement learning, uno dei principali è OpenAI Baselines [17], prodotto da OpenAI, gli stessi che hanno prodotto anche OpenAI Gym. Baselines è una libreria scritta in Python che utilizza come motore per la costruzione

della rete Tensorflow. Racchiude al suo interno l'implementazione di molti algoritmi, i quali possono essere configurati agevolmente e poi allenati per svolgere diversi compiti. Un breve elenco di questi è:

- A2C [20]
- ACER [21]
- PPO [22]
- ACKTR [23]
- DDPG [24]
- DQN [25]
- GAIL [26]
- HER [27]
- TRPO [28]

É particolarmente comodo usare questa libreria insieme a Gym, grazie alla sua diretta compatibilità. Tuttavia, contiene anche diverse problematicità, principalmente collegate ad una carenza di documentazione e di commenti al codice, le quali, poi, comportano anche la difficoltà di modificare il funzionamento degli agenti e di creare un proprio ambiente. Si può descrivere come una scatola nera, con la quale si può rapidamente testare gli ambienti in Gym, però complicata da usare per le proprie necessità. Nonostante le difficoltà, il progetto si basa su di essa, in particolare perché scelta da M. Conciatori che, precedentemente, se ne era servito come base. L'algoritmo adottato da questa libreria è DQN, con tutte le integrazioni presenti, a cui poi sono state applicate le modifiche apportate dal collega che si descriveranno brevemente nei paragrafi successivi.

2.4 Lower Bound DQN

Il lavoro del collega M. Conciatori propone tre estensioni di DQN [2], che sono *Lower Bound DQN*, *Stochastic DQN* ed infine *Entropy DQN* . L'unica estensione su cui ci si soffermerà è Lower Bound DQN, perché è quella da

cui questa tesi prende origine. Lower Bound DQN si discosta da DQN per l'aggiunta di un ulteriore buffer. In questo buffer vengono memorizzati i valori minimi (lower bound) che il Q-value può assumere in uno specifico stato avendo eseguito una determinata azione.

L'importanza del lower bound è rappresentata dal fatto che non è una stima, ma un calcolo esatto, perché viene calcolato alla conclusione dell'episodio sulla base delle azioni svolte dall'agente. I valori contenuti nel lower bound buffer aggiuntivo sono molto simili a quelli del replay buffer, con la differenza che quelli del primo sono calcolati unicamente alla conclusione dell'episodio, l'unico momento in cui si hanno i dati definitivi.

Il calcolo per il lower bound è così composto:

$$LB(s_t, a_t) = \sum_{i=t}^{T} r_i \times \gamma^i \tag{2.3}$$

Si sommano tutti i punteggi, ridotti tramite γ, delle mosse successive, da t fino a quella di conclusione dell'episodio. γ^i rappresenta il fatto che se l'azione che ha portato al reward è lontana rispetto a quella interessata, all'aumentare di i si riduce la ricompensa ricevuta.

Con l'utilizzo del fattore di discount γ, un'azione svolta perde importanza tanto più si trova distante da quella che ha portato ad un reward, e, inoltre, più il valore di γ sarà basso, maggiore sarà la riduzione del reward cumulativo. Se non avvenisse un reward la somma rimarrebbe sempre a 0, per questo motivo i lower bound hanno un'utilità solo negli episodi in cui si ha raggiunto almeno un reward.

Si utilizza il lower bound per dare importanza a quelle azioni che sono state sottostimate dalla rete. Si allena la rete su valori più alti rispetto a quelli stimati, così da farle capire l'importanza di queste azioni ed aumentare la stima precedente. Per l'addestramento vengono selezionati casualmente dal lower bound buffer la metà degli esempi da dare alla rete, cioè 16, se presenti,

e i rimanenti 16 o più sono estratti dal solito replay buffer.

Per memorizzare i dati all'interno del buffer si è applicata una policy per decidere come gestire i record all'interno del lower bound buffer. Non si salvano indistintamente tutti i lower bound ottenuti, ma si effettua un test che ne valuta la bontà, e in base al risultato vengono salvati o meno. Il test in questione consiste nel confrontare il lower bound con il valore stimato per la stessa azione, il Q-value, e nel caso in cui il lower bound sia maggiore di questo, il dato avrà una sua rilevanza per l'addestramento della rete e sarà quindi registrato. Il lower bound, infatti, è utile in quei momenti in cui la rete neurale associa un Q-value eccessivamente basso ad un'azione in un definito stato. Nel momento in cui il Q-value corrispettivo ad un lower bound diventa maggiore, allora, quest'ultimo sarà diventato superfluo e lo si potrà anche eliminare dal buffer. Il test, perciò, viene effettuato anche successivamente all'addestramento per poter controllare se ci fossero dati non più necessari, che quindi vengono rimossi.

2.4.1 Hyperparameters iniziali

Lower Bound DQN	
lr	10^{-4}
replay_batch_size	32
lb_batch_size	16
target_network_update_freq	10^3
gamma	0.99
train_freq	4
buffer_size	10^4
learning_starts	10^4
param_noise	False
prioritized_replay	False
dueling	False
total_timesteps	4×10^6
exploration_fraction	0.1
exploration_final_eps	0.01

Tabella 2.1: Lower Bound DQN Hyperparameters

Nella tabella 2.1 sono elencati gli hyperparameters utilizzati dal collega M. Conciatori con i relativi valori di default. Alcuni tra questi sono già stati visti e descritti all'interno di altri capitoli o paragrafi precedenti, ma si descriveranno comunque brevemente le loro funzioni:

- `lr`: learning rate, il quale controlla l'apporto delle nuove esperienze sui pesi della rete;
- `replay_batch_size`: quantità di esempi estratti dal replay buffer per svolgere l'addestramento. Alla cifra richiesta vanno tolti i record scelti dal lower bound buffer;
- `lb_batch_size`: quantità di esempi estratti dal lower bound buffer. Nel caso in cui il valore sia minore del numero di record richiesti, i restanti sono estratti dal replay buffer;
- `target_network_update_freq`: frequenza con cui aggiornare i pesi della target network con quelli della rete principale;
- `gamma`: il fattore di discount che influisce sul peso assegnato alle ricompense future rispetto a quelle immediate. Impostato al valore 0.99;
- `train_freq`: frequenza con cui svolgere l'addestramento della rete;
- `buffer_size`: dimensione assegnata ad ogni buffer, la dimensione è la stessa per entrambi i buffer. Se uno dei due dovesse riempirsi, l'azione successiva va a sostituirsi a quella più vecchia presente;
- `learning_starts`: parametro per decidere quando iniziare la fase di addestramento. Settato ad un valore uguale alla dimensione del replay buffer, così da iniziare nel momento in cui il replay buffer risulti pieno;
- `param_noise`: booleano per decidere se impiegare la modifica a DQN descritta in [15]

- `prioritized_replay`: booleano per usare o meno la versione prioritized buffer, definita in [10];
- `dueling`: booleano per stabilire se attivare o meno la variante dueling, presentata in [11];
- `total_timesteps`: numero totale di passi da eseguire. Solitamente nell'ordine dei milioni;
- `exploration_fraction`: variabile necessaria per la ε-greedy policy. Indica la percentuale delle mosse e quindi del tempo da dedicare all'esplorazione iniziale rispetto alla quantità totale disponibile;
- `exploration_final_eps`: come la precedente si riferisce alla ε-greedy policy. Gestisce il valore di ε, imponendo il valore minimo da raggiungere, cioè terminata la fase di esplorazione, quale sarà la probabilità che si eseguano ancora mosse casuali.

Target_network_update_freq e *learning_starts* hanno valori indicati in numero di passi.

Capitolo 3

Modifiche apportate

Il reinforcement learning ha ottenuto diversi successi recentemente, ma ci sono ancora diverse sfide da affrontare, come la convergenza in policies ottimali a livello locale e inefficienza degli esempi. La tecnica che si discute in questa tesi si pone come nuova e a sostegno dei modelli già esistenti come autonoma, cioè la predizione della morte nei giochi Atari, in particolare Montezuma's Revenge. La predizione della morte è un argomento difficile da trovare nella letteratura scientifica sul reinforcement learning, questo avviene probabilmente perché si preferisce addestrare l'agente sulle azioni migliori, piuttosto che spendere tempo per cercare di fargli capire quali sono quelle sbagliate rischiando anche di creare problemi alla curva dell'apprendimento. Ma, in certi giochi, le ricompense sono talmente sparse che è necessario trovare un'alternativa.

L'intento è, quindi, di aiutare l'agente a comprendere quali azioni siano controproducenti e portino la sua sconfitta, e, di conseguenza, la morte. É come se l'agente debba predire che alcune mosse sono da evitare, trasformando quelli che prima erano stati terminali, unicamente in transizioni. Spesso le azioni che causano la fine della partita sono per di più quelle più importanti per l'obiettivo da raggiungere, perciò ottenere la capacità di capire quale sia la mossa giusta da fare, anche per esclusione, porta l'agente in una posizio-

ne di vantaggio sull'intera partita, anche perché alcuni ostacoli sono spesso riprodotti più volte all'interno del gioco, imparando a superarli una volta, diventa poi sistematico riuscire a superarli anche tutte le altre. Talvolta, però, le stesse azioni che ci portano alla morte possono anche essere le stesse che ci aiutano a raggiungere la vittoria della partita o la semplificazione del percorso da intraprendere per ottenere il successo, ma questo avviene unicamente nei casi in cui l'agente ha la possibilità di perdere una vita, e da questa perdita ne trae guadagno. In questa tesi non si considerano queste casistiche perché comunque marginali ed in ogni caso cercano di aggirare il problema, come il cercare di evitare un nemico, senza affrontarlo direttamente.

In un contesto come Montezuma's Revenge, l'episodio terminerà unicamente nel caso in cui l'agente compie un'azione che lo porta alla morte. Essendo un gioco con ricompense molto sparse la probabilità di ottenerne una è molto rara, perciò è praticamente impossibile che l'ultima azione eseguita possa essere quella che ha portato risultati positivi all'agente, è bene, quindi, che se ricapitasse nella stessa situazione, cercasse di evitare di eseguire nuovamente lo stesso errore. In un ambiente come il gioco in questione, dove oltre ai reward sparsi, abbiamo anche un gran numero di azioni tra cui scegliere, poterne escludere alcune, diminuirebbe sensitivamente la possibilità di incorrere in errori, garantendo una vita più longeva. É comunque fondamentale non privare l'agente dell'attitudine al rischio, perché questa abilità è il cardine alla base della buona riuscita, senza la quale non si otterrebbero mai dei risultati, ma al massimo si prolungherebbe la vita dell'agente, portando in qualsiasi caso ad un nulla di fatto.

3.1 I primi passi

Il lavoro iniziale era focalizzato su provare ad ottenere dei risultati positivi nel gioco Montezuma's Revenge, questo perché il modello di M. Conciatori non restituiva valori soddisfacenti su questo specifico gioco. Per questa ragione la maggior parte dei cambiamenti che ora andremo a descrivere sono svolti

con la prospettiva di apportare un avanzamento in questo caso particolare. Si è, quindi, puntato ad aiutare l'agente a valorizzare quelle poche o uniche ricompense che riusciva a ricevere.

Per alcuni degli hyperparameters menzionati nel paragrafo precedente sono stati usati diversi valori nelle simulazioni svolte:

- `learning_starts` è stato modificato per far sì che venga gestito direttamente all'interno del gioco, così da iniziare l'addestramento unicamente dopo che sia avvenuta la prima ricompensa, altrimenti sarebbe inutile visto che non ci sono esperienze registrate da cui l'agente possa apprendere. Mentre l'addestramento inizierebbe a 10.000 passi, cioè quando si riempie il buffer, nel caso in cui la prima ricompensa avvenisse prima di quell'istante. Questa modifica ha causato la necessità di una correzione anche alla gestione della variabile *exploration_fraction*, visto che la decrescita di ε inizia unicamente dopo il primo reward;
- `total_timesteps` che gestisce il numero totale di mosse, è stato modificato più volte, ma il valore usato principalmente è stato 4 milioni;
- `gamma` è stato modificato un'unica volta, cambiando il valore da 0.99 a 0.97 per dare più importanza alle ricompense immediate rispetto a quelle future;
- `target_network_update_freq` è stato provato ad aumentare il valore, precedentemente fissato a 10^3, portandolo prima a 5×10^3 e poi a 10^4;
- `train_freq` si occupa della frequenza dell'addestramento, si è di conseguenza provato ad aumentarlo così da diminuirlo moderatamente.

3.1.1 Aumento del reward

La prima modifica svolta per provare a dare più importanza ad azioni lontane nel tempo, è stata aumentare significativamente la quantità di ricompensa assegnata da 1 a 100, visto che il gioco ha ricompense molto sparse, come

già spiegato, è necessario eseguire un numero elevato di azioni prima di poter ottenere il primo reward. Grazie a questo cambiamento tutte le azioni sul cammino che ha portato al reward, anche quelle distanti, potevano ricevere una ricompensa adeguata.

In concomitanza con questa modifica, sono stati cambiati anche i valori di alcuni hyperparameter, *train_freq*, *gamma* e *lb_batch_size*. Con il primo è stata diminuita la frequenza dell'addestramento portandola da 4 a 10, aumentando le esperienze a disposizione del modello.
Il fattore di discount è stato ridotto per non aumentare eccessivamente la propagazione della ricompensa, ora più elevata, sulle azioni molto lontane.
Infine, si è ridimensionato il numero di record estratti dal lower bound buffer portandolo da 16 ad 1, ciò per il sospetto che queste esperienze influissero troppo durante l'addestramento della rete e, quindi, portassero il modello a minimi locali, cioè a focalizzarsi unicamente su quelle azioni e perdere, perciò, la capacità di generalizzare, compiendo dunque decisioni sbagliate.

Le variazioni non hanno portato, però, i risultati attesi. I valori stimati si sono infatti uniformati alla modifica, facendo sì che fosse necessaria una cifra sempre più alta per poter prendere l'azione in considerazione. Nemmeno gli altri cambiamenti hanno portato migliorie, infatti il modello non è comunque riuscito ad ottenere risultati soddisfacenti.

3.1.2 Valore errato Q-values

Agli inizi del progetto il modello non produceva risultati molto positivi, e poco dopo si è notato che i *Q-values*, che venivano passati per effettuare il test per decidere o meno se memorizzare i lower bound, erano inesatti. Riportavano, infatti, valori sempre uguali, anche se le osservazioni del gioco erano diverse, pertanto, si causavano errori quando si svolgeva il test per controllare se inserirli o meno nel lower bound buffer. Di conseguenza, si è deciso di non passare subito i Q-values, ma di passare l'oggetto per calcolarli direttamente dove erano necessari.

3.1.3 Modifica calcolo td_error

Successivamente, si è provato ad aggiornare il calcolo del *td_error*.
Si è sostituita la precedente differenza quadratica tra la rete principale e la rete target, aggiungendo una costante compresa tra 0 ed 1 che andasse a diminuire la differenza tra le due:

$$td\_error = q\_network - \frac{q\_target\_network + c * q\_network}{1 + c}, \text{dove } c \text{ è la costante.} \tag{3.1}$$

L'obiettivo era far sì che se $q\_target\_network < q\_network$, allora, applicando l'equazione di cui sopra, si otterrà che :

$$q\_target\_network < \frac{q\_target\_network + c * q\_network}{1+c} < q\_network.$$

3.2 Passo successivo

Visto che le modifiche svolte non avevano restituito i risultati aspettati, si è pensato che fosse necessario indirizzarsi non più sul cercare di valorizzare i pochi, talvolta anche assenti, reward, ma su un qualcosa di alternativo. Per questa ragione si è considerato che la predizione della morte dell'agente, lo avrebbe aiutato a discernere tra ciò che fosse d'aiuto e ciò che gli avrebbe fatto perdere la partita.

3.2.1 Aggiunta reward negativo

Con l'obiettivo di predire la morte dell'agente si è deciso di aggiungere un reward negativo. Questa penalità si applica per quelle azioni che portano l'agente alla morte. La scelta è stata fatta con l'obiettivo di allenare l'agente a non ripetere successivamente quella azione che l'ha, poi, portato a perdere. Il reward negativo è, quindi, stato assegnato a tutte quelle azioni di conclusione dell'episodio, questo perché se l'episodio termina significa che, particolarmente nel caso in questione, l'agente è morto. Il record viene memorizzato, con le stesse modalità precedentemente descritte, nel replay buffer con un

valore di reward corrispondente a -1. In seguito, per rendere la ricompensa negativa meno influente ai fini dell'apprendimento, si è ridotto il valore a -0.1.

3.2.2 Done buffer

Con l'introduzione della penalità, non solo non si è osservato un miglioramento, ma quasi un peggioramento. Questo è stato causato dal fatto che tra le esperienze estratte dal buffer, quelle con la ricompensa negativa si sceglievano spesso ed il loro peso aumentava con il tempo. Il continuo apporto di sanzioni, portavano l'agente a prediligere un comportamento prudente, cercando, di conseguenza, di evitare il rischio causato dal timore di morire, senza quindi ottenere nessun risultato. Per questo si è pensato, non che l'idea della penalità fosse sbagliata, ma più che altro che ne andasse limitata la quantità. Ciò, anche perché è giusto che l'agente si focalizzi principalmente sulle ricompense positive, quelle negative vanno unicamente considerate secondarie.

Si è, perciò, pensato di eliminare i reward negativi dal replay buffer ed aggiungere un ulteriore buffer in cui immagazzinare unicamente le azioni che portavano alla fine della partita, quindi, con ricompensa negativa, chiamato Done buffer.

I record memorizzati all'interno sono della stessa forma degli altri due buffer, l'unica differenza risiede nel fatto che si aggiunge una sola esperienza per episodio, ovviamente, perché ci sarà solamente un'azione che porta alla conclusione. Il buffer complementare contribuisce solo in minima parte all'apprendimento dell'agente, visto che se ne utilizza solo un unico esempio, in confronto al numero molto più alto estratto dal replay buffer.

Come si descriverà successivamente nel capitolo dei risultati, l'esito ha generato un successo, seppur circoscritto a quei giochi, tra cui Montezuma's Revenge, con ricompense molto sparse, i quali altrimenti avrebbero avuto maggiori problematiche nell'ottenere il primo reward. In altri casi, si nota

anche una piccola flessione rispetto al traguardo raggiunto dal collega. Dopo una certa soglia, che coincide con l'aumento delle ricompense positive, l'aggiunta di una penalità rischia di ridurre la loro l'importanza, causando da parte dell'agente una scelta sbagliata rispetto all'azione migliore da eseguire.

3.3 Tecnologie utilizzate

3.3.1 Colab

A causa della potenza di calcolo e della memoria ram necessaria, ma anche della mole di test e simulazioni previste, ho deciso di utilizzare una piattaforma cloud per eseguire il progetto.

La scelta iniziale è ricaduta su Colaboratory (Colab) [18] ed è stata anche influenzata dal fatto che questo strumento è stato utilizzato dal collega da cui ho ereditato il progetto. Questo servizio è offerto gratuitamente da Google, e, seppur con ovvie limitazioni, permette di eseguire il proprio codice, offrendo la potenza di calcolo e lo spazio necessario per salvare i propri file.

Colab sfrutta l'utilizzo dei Jupyter Notebook, che non sono altro che documenti interattivi nei quali è possibile scrivere e direttamente eseguire il codice. Grazie a questa tecnologia si può suddividere il lavoro in celle, ognuna delle quali può contenere anche del testo informativo che descrive ciò che si sta per svolgere.
Inoltre, per eseguire il calcolo questa piattaforma offre la possibilità di servirsi di GPU o TPU, tra cui le seconde sono specifiche per il calcolo con reti neurali.

Ovviamente questo servizio presenta anche dei difetti. In primis la durata massima delle simulazioni, che è limitata a circa 12 ore, e soprattutto il fatto che al termine di questo arco temporale non vengono salvati i file che si erano caricati ed è quindi necessario ogni volta ricaricarli e reinstallare tutte

le librerie necessarie. L'unica cosa che rimane salvata è il file scritto in primo piano con le sue celle, dove, tuttavia, ognuna dovrà essere rieseguita.
In aggiunta, se allo scadere del tempo il processo non è ancora terminato, può capitare di non poterlo portare a termine e di perdere tutti i risultati ottenuti.

3.3.2 AWS (Amazon Web Services)

Viste le difficoltà riscontrate con l'utilizzo di Colab, si è scelto di usare un servizio più comodo, e che garantisse di non perdere nessun risultato, pertanto la piattaforma iniziale è stata sostituita con AWS (Amazon Web Services).

AWS è un cloud di Amazon che offre la possibilità di usufruire di server con tutta la potenza di calcolo e ram opportuna. A differenza di Colab è a pagamento ed il costo è rapportato alla potenza richiesta, permette però a tutti gli studenti di avvalersi di un certo quantitativo di credito gratuito solo su certe macchine.

In confronto al primo non offre la possibilità di eseguire direttamente il codice, ma è necessario connettersi ed eseguire il programma tramite il servizio *ssh*. Al momento della creazione dell'istanza devono essere installate tutte le librerie necessarie e l'istanza rimarrà in esecuzione fino a che non verrà fermata. Si procede poi a scaricare i files di log con i risultati ottenuti per analizzarli.

3.3.3 Tensorboard

Tensorboard [19] è un strumento messo a disposizione dagli sviluppatori di Tensorflow che permette di analizzare in modo dettagliato la struttura del grafo di computazione, in particolare nel caso in cui la rete sia grande ed il modello complesso.

Tensorboard consente di visualizzare tutte le statistiche relative ad una rete neurale, come i parametri di training (loss, accuracy e pesi), le immagini ed anche il grafo completo. Questo può essere molto utile per capire come si muovono le informazioni all'interno del grafo e quindi fare agevolmente debugging ed ottimizzare il modello.

L'utilizzo di Tensorboard è molto semplificato, è sufficiente creare un oggetto FileWriter a cui indicare la cartella dove salvare i file di log che crea. In seguito, tramite le funzioni dedicate, in particolare Scalar ed Histogram, si stampano rispettivamente valori scalari o grafici che rappresentano la distribuzione dei valori, basandosi su un passo che gli viene assegnato.

Capitolo 4

Risultati ottenuti

In questo capitolo si mostrano i risultati ottenuti, confrontando le perfomance sia con l'algoritmo di M. Conciatori da cui si era partiti, Lower Bound DQN, che con l'algoritmo di base per entrambi, che è DQN. Le simulazioni sono tutte della durata di 4 milioni di passi, questo per garantire uniformità rispetto ai test del collega, ma anche per motivazioni temporali e computazionali. Gli hyperparameters utilizzati sono principalmente quelli di default, per lo stesso motivo già detto, ovvero quello di garantire omogeneità con i test precedentemente svolti. Le poche variazioni agli hyperparameters svolte agli inizi del progetto, che si sono descritte nel capitolo precedente, sono state messe in disparte preferendo i valori di default, grazie ai quali si è potuto mostrare maggiormente le differenze ricollegabili ai cambiamenti eseguiti.

Come citato nell'elaborato del collega M. Conciatori [2], normalmente i risultati sono normalizzati e trasformati in percentuali, dove il 100% indica il punteggio ottenuto da un esperto uman, seguendo una formula precisa. Si è pensato, però, che fosse più uniforme mostrare i risultati nella stessa modalità del collega, così da poter fare un confronto diretto, cioè mostrando i punteggi direttamente estratti dalle simulazioni.

Rispetto ai test svolti dal collega, l'unica differenza è la dimensione di *lb_batch_size*,

il quale è passato da 16 unità a 4, questo per limitare di poco il suo peso. Si mostrano ora i risultati finali di tutti i giochi testati:

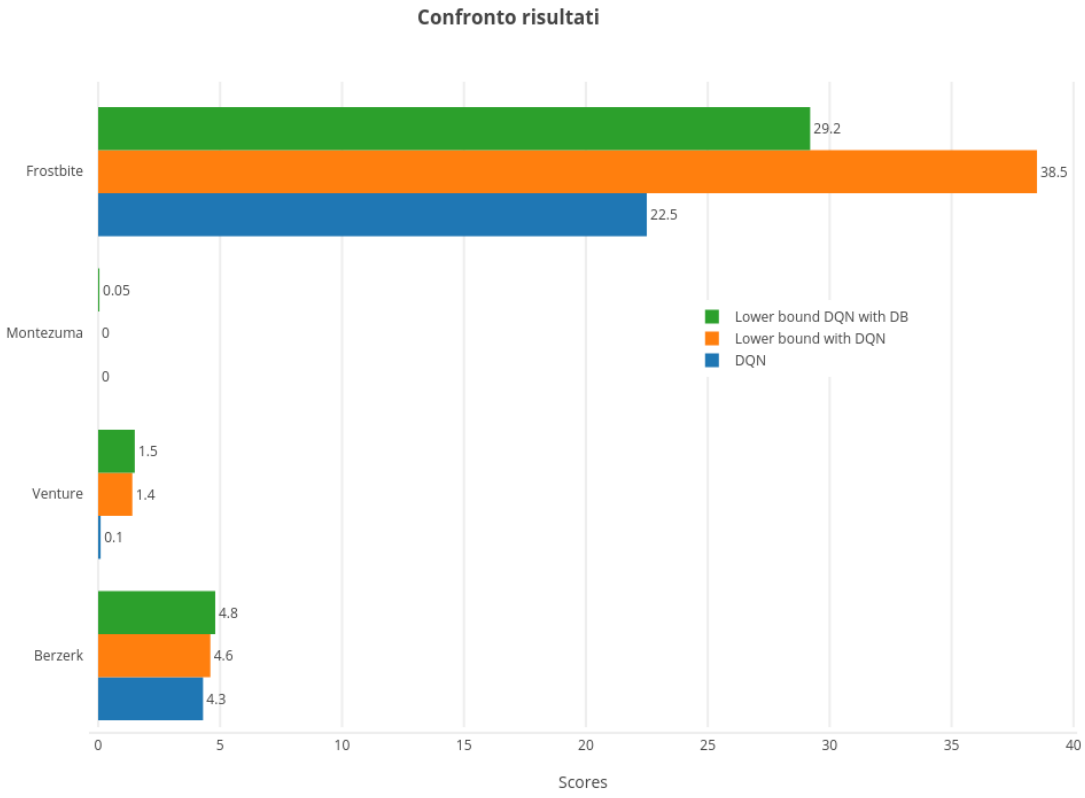

Figura 4.1: Confronto di DQN, Lower Bound DQN e Lower Bound DQN with Done Buffer su diversi giochi.

Si confrontano 4 giochi, Montezuma's Revenge, ormai ampiamente discusso, Frostbite, Venture e Berzerk. Per tutti e quattro il passo utilizzato è stato di 4 milioni di passi. Gli ultimi tre appartengono tutti alla categoria di giochi con ricompense sparse, dove, però, il primo ed il terzo hanno prodotto dei punteggi relativamente soddisfacenti sia in Lower Bound DQN che in DQN, mentre il secondo, così come Montezuma's Revenge, in DQN ha ottenuto un esiguo risultato, raggiungendo un unico reward. Come si può notare Lower Bound DQN ha riportato degli esiti più che accettabili in Frostbite, ma è riuscito ad ottenere risultati consistenti anche in Venture e di poco superiori a

DQN in Berzerk. Al contrario, per quanto riguarda l'algoritmo con le nostre modifiche, il miglioramento non è generale, ma bensì ristretto a quei giochi con risultati più sparsi. Come si può notare i risultati sono contrastanti, non sempre sono migliori di Lower Bound DQN, la motivazione è da ricercare nella tipologia di gioco, è infatti necessario analizzare il gioco per capire se questa tecnica porterà dei vantaggi oppure no. Questo dimostra come spesso sia imprescindibile concentrarsi su un gruppo ristretto di problemi a cui applicare specifici cambiamenti, garantendo di poter ottenere risultati migliori. Ciò non significa che la capacità di generalizzare di un algoritmo non sia importante, anzi è fondamentale, ma se si vuole eccellere è doveroso approfondire le motivazioni per cui non si ottengono dei successi e concentrarsi unicamente sul singolo problema.

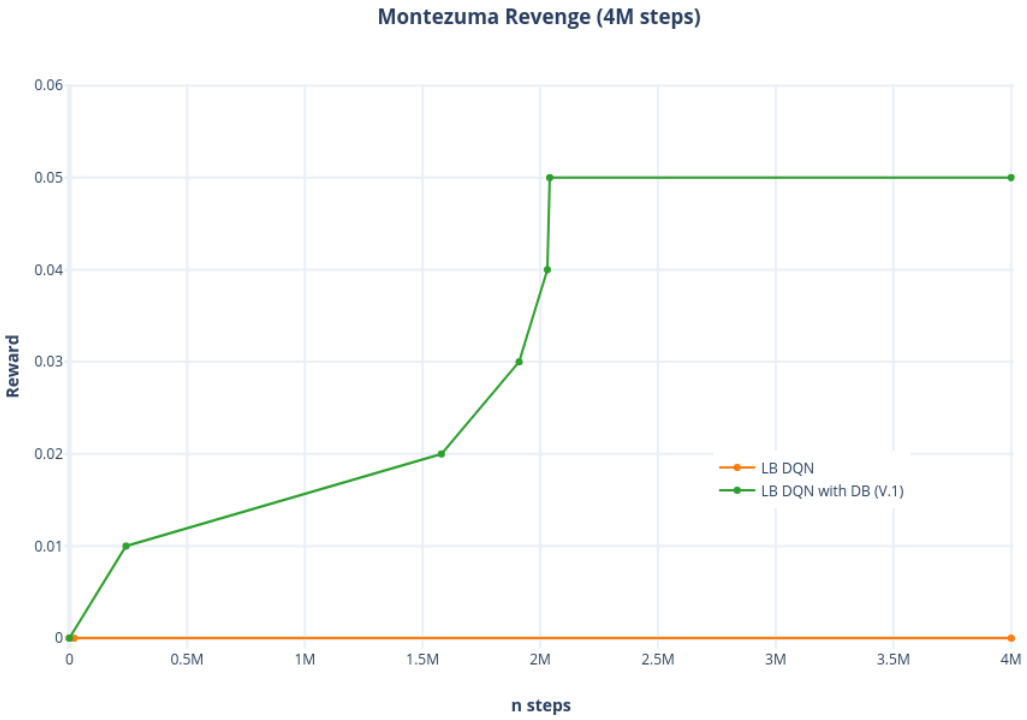

Figura 4.2: Confronto dei risultati di Lower Bound DQN e di Lower Bound DQN with Done Buffer sul gioco Montezuma's Revenge.

Possiamo vedere che in Montezuma, dove l'algoritmo del collega non riusciva

a spostarsi dallo 0, l'aggiunta del done buffer garantisce, oltre che il raggiungimento del reward, una ripetizione ed anche un miglioramento, riuscendo a raggiungere un punteggio, anche se ancora molto basso, di una certa costanza. Non si è mostrata nel grafico la curva collegata a DQN, questo perché il risultato è il medesimo di Lower Bound DQN, cioè rimane stabile sullo 0, senza mostrare nessuna variazione. Si suppone che se il numero di passi fosse stato maggiore, l'algoritmo avrebbe ottenuto maggiore costanza nel trovare il successo, e che questo avrebbe portato a punteggi considerevolmente più alti. C'è comunque da osservare che non si può considerare questo risultato un successo perché non vede una netta miglioria, ma semplicemente un circoscritto aumento.

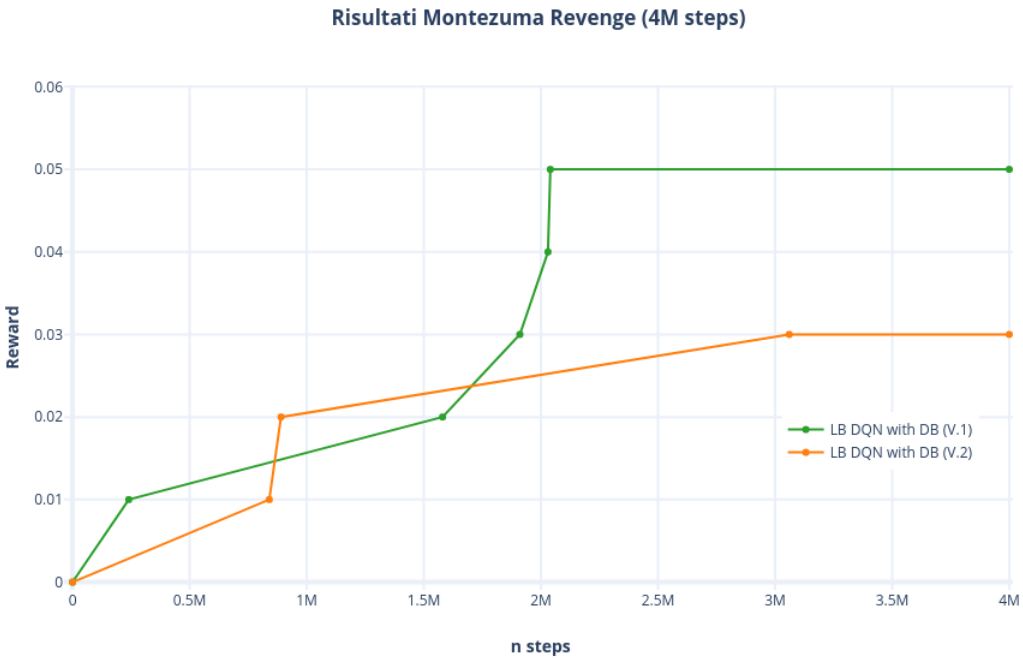

Figura 4.3: Confronto due esecuzioni Montezuma.

Nel confronto tra due esecuzioni di Montezuma si può notare che entrambi ottengono il primo reward entro il primo milione di passi, uno dei due addirittura prima dei 300.000 passi. Si suppone che questo sia garantito dalla combinazione della riduzione dell'esplorazione, insieme all'allenamento rea-

lizzato attraverso le esperienze estratte dal Done Buffer. Il migliore tra i due, anche se la differenza è minima, ha un avanzamento abbastanza repentino in confronto al secondo e questo gli garantisce di ottenere risultati migliori, mentre l'altro rimane stabile su risultati minimi per poi ottenere un piccolo avanzamento. Questo grafico è l'effettiva dimostrazione che sarebbe stato necessario avere un intervallo più grande per poter notare le vere differenze tra le due esecuzioni ed anche per osservare se in un periodo più ampio avrebbero ottenuto, magari, risultati simili. Con questa tipologia di giochi i tempi si dilatano notevolmente perché si rende necessario lasciare riservato all'agente un periodo considerevole per esplorare il nuovo ambiente, senza il quale l'esito sarebbe infruttuoso.

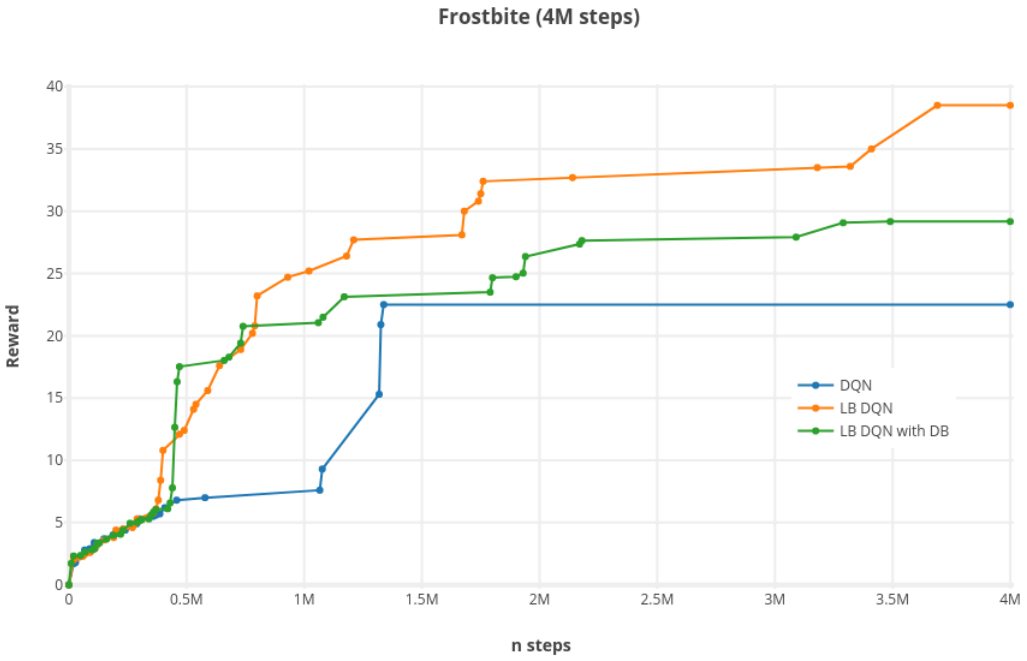

Figura 4.4: Confronto dei risultati dei tre algoritmi sul gioco Frostbite.

Infine si mostrano gli andamenti dei tre algoritmi confrontati sul gioco Frostbite. Come si vede nel grafico, il risultato dell'algoritmo proposto dimostra nella parte iniziale un andamento simile a Lower Bound DQN, ed entrambi si discostano nettamente da DQN, che rimane fisso dopo un certo numero

di passi. I primi due, inoltre, presentano un persistente aumento dei valori nel tempo, continuando a crescere, anche se sempre più lentamente, mentre DQN si arresta e raggiunge un plateau su un preciso valore prima di 1 milione e mezzo di passi.

In tutti i risultati c'è da considerare che nel punteggio è compreso il reward negativo, che riduce, anche se solo marginalmente, il valore finale. Più basso è il punteggio, maggiore ne sarà l'incidenza. Da questo risultato si evince che il reward negativo ha un apporto positivo solo in quei giochi dove l'ottenimento della ricompensa è molto sparso, altrimenti riduce unicamente l'apporto che potrebbe dare la ricompensa positiva. Quindi, non è né utile né funzionale aggiungere il reward negativo nel caso in cui il modello riesca ad apprendere ugualmente senza il suo impiego. C'è anche da considerare che all'aumentare del tempo, l'utilizzo delle penalità non aiuta l'apprendimento, ma sono d'aiuto nel momento iniziale per far avviare l'apprendimento dell'algoritmo.

Capitolo 5

Difficoltà riscontrate

Nello svolgere questo lavoro si sono presentate diverse problematiche. La prima è legata al fatto che l'algoritmo fornitomi dal mio collega restituiva degli errori e quindi non procedeva all'esecuzione. Ho, perciò, dovuto spendere una considerevole quantità di tempo per riuscire a risolvere il problema, il quale era stato in principio attribuito a versioni diverse di file che erano stati aggiornati nel framework OpenAI baselines, insieme alla versione della libreria di computazione numerica Tensorflow. Successivamente ho poi scoperto che le problematicità collegate al software di OpenAI, erano in realtà da associare all'altra libreria da loro sviluppata, e cioè Gym.

Un'altra limitazione, principalmente temporale, è dovuta all'utilizzo di Colaboratory, infatti è possibile utilizzare il servizio unicamente per 12 ore, con il rischio, come già detto, che si possa perdere i risultati ottenuti o che l'esecuzione non sia ancora terminata. Successivamente l'impiego di AWS ha risolto il principale aspetto legato al vincolo di durata, vista l'assenza di limiti, ci si è, però scontrati con un altro problema, ossia la disponibilità di risorse, che per gli studenti è molto ridotta, e questo ha prolungato le tempistiche per ogni singola esecuzione. L'unica soluzione che si è considerata relativamente soddisfacente è stata la combinazione dei due servizi, avendo, di conseguenza, la possibilità di eseguire più di una volta il modello.

La difficoltà principale è da rapportare alle tempistiche, infatti per ogni esecuzione e per avere a disposizione i risultati è necessario molto tempo, soprattutto in un gioco come Montezuma, dove sono necessari, talvolta, anche milioni di passi spesi a far esplorare l'ambiente all'agente per ottenere la prima ricompensa. Questo ha comportato notevoli ritardi rispetto ai cambiamenti eseguiti, oltre che una diminuzione del numero di passi eseguibili, impostato in modo fisso su 4 milioni. Probabilmente l'allungamento dei passi d'esecuzione comporterebbe anche un miglioramento dei risultati, in particolare su giochi a ricompense molto sparse dove è necessario più tempo da dedicare all'esplorazione.

L'ultima complicazione risiede proprio nella libreria OpenAI baselines, la quale ha un codice poco chiaro, che fa da ostacolo rispetto a modifiche da parte di terzi e sarebbe, dunque, necessario un refactoring completo del codice che lo renda più leggibile, proprio a causa di questo è nata una versione più stabile, chiamata proprio Stable Baselines [29].

Capitolo 6

Conclusioni

In questo elaborato si è mostrata un'introduzione al Reinforcement Learning, aggiungendo un argomento poco trattato, cioè la predizione della morte dell'agente. Si è illustrato l'uso delle reti neurali applicato al RL, in particolare ci si è focalizzati sull'algoritmo DQN. Si è voluto migliorare un algoritmo, Lower Bound DQN, che già produceva ottimi risultati, per cercare di raggiungere un incremento laddove mostrava delle carenze. Come si può evincere dai risultati ottenuti, molti dei modelli dimostrano, ancora, di ottenere risultati molto ridotti rispetto agli obiettivi preposti. C'è anche da considerare che per conseguire questi risultati è necessario che l'algoritmo svolga milioni di passi, spendendo un tempo maggiore di quanto ne spenderebbe un giocatore umano.

6.1 Sviluppi futuri

Alcuni miglioramenti che si potrebbero applicare sono:

- Predire l'intera sequenza di azioni che porta alla morte dell'agente, cioè prevedere qual è la mossa che inizia la serie che poi conduce alla terminazione, ampliando la ricompensa negativa a tutta la catena;

- É chiaramente più importante puntare sulle serie di azioni che generano ricompense positive, perciò si potrebbero escludere le penalità nel momento in cui sia stato raggiunto un numero minimo di successi, così da garantire che le penalità non limitino l'apprendimento;
- Comprendere quali stati siano più importanti rispetto ad altri, cioè quelli in cui prendere un'azione a differenza di un'altra porta successi o fallimenti, e favorire l'addestramento dell'agente maggiormente su questi precisi momenti;
- Imparare ogni singola sequenza di azioni positiva che nel gioco tende a ripetersi, per esempio riguardo a Montezuma imparare a scendere la scala o saltare la corda, riconoscendo delle ricompense intermedie, oltre a quelle già fissate.

Bibliografia

[1] Richard S. Sutton and Andrew G. Barto, 2018.
"Reinforcement Learning: An Introduction"
The MIT Press (2nd edition).

[2] M. Conciatori. 2019.
"Tecniche di deep learning applicate a giochi Atari".
AMSLaurea, AlmaDL: University of Bologna Digital Library.

[3] D. Silver. UCL Course on RL,
`http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`.

[4] Marc G. Bellemare, Yavar Naddaf, Joel Veness, Michael Bowling.
"The Arcade Learning Environment:An Evaluation Platform for General Agents". ArXiv preprint arXiv:1207.4708.

[5] OpenAI Gym.
`https://gym.openai.com/`.

[6] C.J.C.H. Watkins, P. Dayan.
"Q-learning"
Mach Learn 8, 279–292 (1992).

[7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmille.
"Playing Atari with Deep Reinforcement Learning". ArXiv preprint arXiv:1312.5602.

[8] Arthur L Samuel.
"Some studies in machine learning using the game of checkers. II - recent progress". In: Computer Games I. Springer, 1988, pp. 366–400.

[9] Max Jaderberg et al.
"Human-level performance in 3D multiplayer games with population-based reinforcement learning".
In: Science 364.6443 (mag. 2019). issn: 1095-9203. doi: 10.1126/science.aau6249. http://dx.doi.org/10.1126/science.aau6249.

[10] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver.
"Prioritized experience replay". ArXiv preprint arXiv:1511.05952.

[11] ZiyuWang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot. *"Dueling Network Architectures for Deep Reinforcement Learning". ArXiv preprint arXiv:1511.06581.*

[12] N. Justesen et al.
"Deep Learning for Video Game Playing".
In: IEEE Transactions on Games (2019). issn: 2475-1502. doi: 10.1109/TG. 2019.2896986.

[13] Marc Bellemare et al.
"Unifying count-based exploration and intrinsic motivation".
In: Advances in Neural Information Processing Systems. 2016, pp. 1471–1479.

[14] Volodymyr Mnih et al.
"Human-level control through deep reinforcement learning".
In: Nature 518.7540 (2015), p. 529.

[15] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, Marcin Andrychowicz.
"Parameter Space Noise for Exploration". ArXiv preprint arXiv:1706.01905.

[16] Greg Brockman et al.
"Openai gym".
In: arXiv preprint arXiv:1606.01540 (2016).

[17] OpenaAI Baselines.
`https://github.com/openai/baselines`.

[18] colab.research.google.com. Welcome to Colaboratory.
`https://colab.research.google.com/notebooks/welcome.ipynb`.

[19] D Man´e et al.
"TensorBoard: TensorFlow's visualization toolkit", 2015.

[20] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves,Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu.
"Asynchronous methods for deep reinforcement learning. In International conference on machine learning". pages 1928-1937, 2016.

[21] Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos,Koray Kavukcuoglu, and Nando de Freitas.
"Sample effcient actor-critic with experience replay".arXiv preprint arXiv:1611.01224, 2016.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *"Proximal policy optimization algorithms". arXiv preprint arXiv:1707.06347*, 2017

[23] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. *"Scalable trust-region method for deep reinforcement learning using kroneckerfactored approximation".*
In Advances in neural information processing systems, pages 5279-5288, 2017.

[24] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra.

"Continuous control with deep reinforcement learning". arXiv preprint arXiv:1509.02971, 2015

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness,Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland,Georg Ostrovski, et al.
"Human-level control through deep reinforcement learning".
Nature, 518(7540):529, 2015.

[26] Jonathan Ho and Stefano Ermon.
"Generative adversarial imitation learning".
In Advances in Neural Information Processing Systems, pages 4565-4573,2016.

[27] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba.
"Hindsight experience replay".
In Advances in Neural Information Processing Systems, pages 5048-5058, 2017.

[28] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and PhilippMoritz.
"Trust region policy optimization".
In Icml, volume 37, pages 1889-1897, 2015.

[29] Ashley Hill et al. Stable Baselines.
`https://stable-baselines.readthedocs.io/en/master/`

Ringraziamenti

Un ringraziamento particolare va al professor Andrea Asperti, che mi ha accompagnato durante tutto il progetto. Mi sembra, inoltre, doveroso esprimere gratitudine al collega Marco Conciatori, che ha risposto a tutti i quesiti che avevo sul suo progetto e si è sempre reso disponibile.
Ringrazio tutti quelli che mi sono stati vicini, tra cui, in particolare, vorrei soffermarmi su Andia, la mia ragazza che mi ha supportato e sopportato, e tutti gli amici, sia quelli conosciuti in questi ultimi tre anni, in particolare durante il periodo Erasmus, che quelli di sempre. Un ringraziamento è dovuto anche alla mia famiglia, che mi ha accompagnato in questo percorso e supportato sia emotivamente che economicamente.
Infine, ringrazio l'università di Bologna per il percorso che mi ha permesso di intraprendere ed in particolare per aver reso possibile la mia esperienza Erasmus, che è stata oltre che una bellissima avventura, anche un'attività formativa.