Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

TECNICHE DI DEEP LEARNING APPLICATE A GIOCHI ATARI

Relatore:
Chiar.mo Prof.
ANDREA ASPERTI

Presentata da:
MARCO CONCIATORI

II Sessione
Anno Accademico 2018-2019

Introduzione

Il contesto di riferimento in cui situare gli argomenti trattati nel corpo della tesi è rappresentato dal cosiddetto *reinforcement learning*. Con questo termine si indica un particolare settore dell'intelligenza artificiale che affronta problemi complessi, non approcciabili con altre tecniche di programmazione. Gli algoritmi di reinforcement learning, detti anche agenti, apprendono tramite interazioni con l'ambiente attraverso "ricompense" e "punizioni" in funzione del vantaggio ottenuto o meno con le varie azioni compiute.

La massimizzazione delle ricompense condiziona così l'agente all'ottenimento delle migliori sequenze di azioni (strategie), anche partendo da una base conoscitiva nulla e senza alcun intervento umano.

In questo ambito gli algoritmi - agenti - ormai di uso comune sono basati sulle reti neurali (deep neural network) che simulano le modalità di apprendimento tipiche degli esseri umani e di altri animali, ispirandosi ai processi cognitivi descritti dalle neuroscienze [20].

Tali modalità di apprendimento autonome sono proprio quelle che consentono all'agente di affrontare problemi dinamici che richiedono sequenze di decisioni appropriate nel tempo nonché capacità previsionale (ogni decisione avrà infatti conseguenze nello sviluppo futuro dell'ambiente).

Nel concreto, esempi di problemi affrontati con le tecniche di reinforcement learning sono gli scacchi e la dama, appartenenti alla categoria dei giochi deterministici ad informazione completa, oppure videogiochi di vario genere e difficoltà come Doom (caratterizzato da ambiente 3D, stati parzialmente osservabili, real time). Si tratta comunque di ambienti e problemi

rientranti, dal punto di vista teorico, nel modello markov decision process (vedi FMDP).

Nel quadro di riferimento appena enunciato, la presente tesi sperimentale è stata sviluppata intorno all'idea di modificare uno specifico algoritmo, già utilizzato per affrontare problemi di reinforcement learning con reti neurali, per migliorarne le performance. L'algoritmo su cui è incentrata la ricerca è Deep Q-Network (DQN), storicamente uno dei primi ad essere stati implementati e che, sfruttando le deep neural network, è riuscito a produrre risultati significativi.

Esiste tuttavia un sottoinsieme di problemi in cui gli algoritmi di reinforcement learning, tra cui lo stesso DQN, incontrano difficoltà nel conseguimento di risultati soddisfacenti: si tratta dei cosiddetti problemi a ricompense sparse. La principale difficoltà insita in tali ambienti è rappresentata dall'elevatissimo numero di azioni che l'agente deve compiere per raggiungere solo uno scarso numero di ricompense, fatto - questo - a sua volta legato al mantenimento di un basso livello di apprendimento, che genera quindi tempi di risposta troppo elevati per la costruzione di una strategia efficace.

Le modifiche introdotte sull'algoritmo DQN hanno lo scopo di migliorarne le modalità di apprendimento proprio nel contesto dei problemi a ricompense sparse, seguendo, in estrema sintesi, due principali direzioni: un miglior sfruttamento delle poche ricompense a disposizione attraverso un loro più frequente utilizzo (variante "Lower Bound DQN", sezione 3.1) oppure una esplorazione più efficace (varianti "Massima esplorazione fino alla prima ricompensa" alla sezione 3.2, "Stochastic DQN" alla sezione 3.4, "Entropy DQN" alla sezione 3.5).

Complessivamente le varianti realizzate mostrano miglioramenti anche significativi nei test di funzionamento e confronto ("Test e Risultati", Capitolo 5) e, pur rilevando nei risultati una varianza elevata in funzione dell'ambiente di test, essi non scendono mai al di sotto dei livelli raggiunti dall'algoritmo originale.

Di seguito si riporta una panoramica della struttura della tesi che illustra

i principali argomenti trattati.

Il primo capitolo è dedicato ad un inquadramento generale dell'oggetto di studio, rivolgendosi in particolare agli ambiti di machine learning e reinforcement learning.

Nel secondo capitolo viene trattato specificamente l'algoritmo fondamentale, che ha rappresentato anche il punto di partenza del lavoro: Deep Q-Network, del quale vengono presentate le generalità del funzionamento e le varie versioni esistenti.

Nel terzo capitolo si passa poi all'esposizione del nucleo della tesi, cioè la parte sperimentale da me realizzata. In particolare durante il lavoro preliminare mi sono dedicato all'implementazione di una sola modifica di DQN denominata Lower Bound DQN; successivamente, per ampliare la sperimentazione, ho introdotto ulteriori modifiche all'algoritmo (tra le quali vale la pena anticipare Stochastic DQN, sezione 3.4 ed Entropy DQN, sezione 3.5).

Nel quarto capitolo vengono illustrate le motivazioni alla base della scelta di utilizzare come banco di prova i giochi Atari 2600: fondamentalmente il fatto che essi siano diventati uno standard in questo campo di ricerca (per numerosità di ambienti disponibili, diversità di generi e livelli di difficoltà). Viene anche esposto l'utilizzo di OpenAI Gym, come toolkit che implementa i giochi Atari.

Il quinto capitolo presenta i risultati dei vari test e allenamenti compiuti, corredando il testo con le rappresentazioni grafiche più significative. Questa sintesi permette di valutare l'efficacia delle varianti realizzate: a tal fine nella fase di sperimentazione sono stati eseguiti numerosi test che combinassero in svariati modi le suddette modifiche contemplando anche variazioni degli iperparametri già presenti in DQN. Infine ho elaborato i risultati ottenuti comparandoli con le performance dell'algoritmo originale e di preesistenti versioni modificate di DQN. Dal raffronto sono state così individuate le modifiche che consentono il raggiungimento dei risultati migliori, cioè superiori alla baseline (rappresentata dai punteggi conseguiti da DQN originale).

Il capitolo 6 raccoglie in sintesi le principali difficoltà riscontrate nel corso

della sperimentazione, sia quelle legate agli applicativi esterni utilizzati, sia quelle relative alla realizzazione delle stesse varianti.

Il capitolo finale contiene una valutazione personale dell'esperienza accompagnata da alcune indicazioni di potenziali direzioni di sviluppo.

Indice

Introduzione **i**

Elenco delle figure **viii**

Elenco delle tabelle **ix**

1 Background **1**

1.1 Machine Learning 1

1.2 Reinforcement Learning 2

2 DQN Originale **5**

2.1 Q-Learning 5

2.2 DQN 6

2.3 Versioni preesistenti di DQN 12

3 Modifiche a DQN **15**

3.1 Lower Bound DQN 15

3.2 Massima esplorazione fino alla prima ricompensa 18

3.3 Salvataggio e Caricamento di un addestramento interrotto . . 19

3.4 Stochastic DQN 21

3.5 Entropy DQN 23

3.6 Note su iperparametri e modifiche ad un loro sottoinsieme . . 25

3.7 Tecnologie utilizzate 30

3.7.1 TensorBoard 31

4 Atari 2600 **33**
4.1 OpenAI Gym 34

5 Test e Risultati **37**
5.1 Lower Bound DQN 38
5.2 Stochastic DQN 46
5.3 Entropy DQN 47

6 Difficoltà incontrate **51**

Conclusioni **53**
6.1 Sviluppi futuri 53

A Definizioni e altri termini di frequente utilizzo **57**

Bibliografia **58**

Elenco delle figure

1.1 Schema di funzionamento astratto di un generico problema di RL. 4

5.1 Risultati ottenuti da LB DQN e DQN originale; il numero tra parentesi indica quanti milioni di step sono stati usati per ogni addestramento. 39

5.2 LB DQN (arancione) e DQN originale (blu) a confronto in Frostbite (Atari 2600) 40

5.3 LB DQN (arancione) e DQN originale (blu) a confronto in Venture (Atari 2600). Simulazione da 2M di passi. 41

5.4 LB DQN (arancione) e DQN originale (blu) a confronto in Venture (Atari 2600). Simulazione da 4M di passi. 41

5.5 Risultati ottenuti da LB Dueling DQN e Dueling DQN; il numero tra parentesi indica quanti milioni di step sono stati usati per ogni addestramento. 42

5.6 Risultati ottenuti da LB DQN e DQN originale con combinazioni di Parameter Space Noise e Dueling attive. Tutte le simulazioni usate sono ottenute con $1M$ di step di addestramento. 43

5.7 LB DQN e DQN originale e Prioritized Replay a confronto. Le simulazioni usate sono ottenute con $4M$ di step di addestramento su Frostbite. 44

5.8 Risultati ottenuti da LB DQN in Frostbite con simulazione da $20M$ di step. 45
5.9 Risultati ottenuti da Stochastic DQN in Frostbite con simulazioni da $4M$ di step. 46
5.10 Comparazione fra Stochastic DQN, LB DQN e Original DQN in Frostbite con simulazioni da $4M$ di step. 47
5.11 Confronto fra le due varianti di Entropy DQN. Simulazioni in Frostbite con $4M$ di step. 48
5.12 Confronto diretto fra tutte le modifiche implementate e DQN originale. Simulazioni in Frostbite con $4M$ di step. 49

Elenco delle tabelle

3.1 Iperparametri di base utilizzati in tutte le versioni di DQN (salvo variazioni esplicite). 26
3.2 Differenze fra gli iperparametri generali e quelli di Lower Bound DQN. 27
3.3 Differenze fra gli iperparametri generali e quelli di Stochastic DQN. 27
3.4 Differenze fra gli iperparametri generali e quelli di Entropy DQN. 28

Capitolo 1

Background

1.1 Machine Learning

Per introdurre la branca del machine learning occorre iniziare dal più generale concetto di intelligenza artificiale che, considerando le varie declinazioni reperibili attualmente, si potrebbe così definire: abilità di un sistema di interpretare correttamente dati provenienti da fonti esterne, imparare da questi dati e utilizzare le conoscenze apprese per raggiungere obbiettivi specifici in modo flessibile (v. in particolare [15]).

L'espressione machine learning coniata inizialmente da Arthur Samuel [27] è stata successivamente formalizzata da Tom M. Mitchell nel seguente enunciato: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E." [18].

Machine learning si inquadra dunque come un ramo di intelligenza artificiale che studia algoritmi in grado di portare a termine compiti specifici senza avere ricevuto esplicitamente le istruzioni relative, tramite inferenza e con l'utilizzo di dati di allenamento.

Machine learning si può dividere in alcune categorie principali.

- Supervised learning: l'algoritmo impara ad associare ad ogni input l'output corretto tramite labeled data (dati di allenamento composti

da coppie input-output). Esempi classici sono rappresentati da attività di *image recognition* e *spam filtering*.

- Unsupervised learning: in questo caso i dati di training contengono solo l'input, e non il corrispondente output. Nel contesto di unsupervised learning, l'algoritmo cerca pattern e similitudini fra i dati forniti, scegliendo in autonomia i criteri di raggruppamento. Tali metodi vengono utilizzati per scoprire features comuni a certi gruppi di dati, o eseguire clustering.

- Active learning: in questo caso l'algoritmo ha la possibilità di conoscere il label (output corrispondente) solo di alcuni dei dati forniti, e deve scegliere quindi quelli che massimizzano l'informazione guadagnata.

- Reinforcement learning: il problema si compone di un ambiente e un agente. L'agente interagisce con l'ambiente compiendo azioni, l'ambiente risponde fornendo una ricompensa e l'osservazione del proprio nuovo stato all'agente. L'agente, basandosi sulle ricompense ricevute, modifica il proprio comportamento in modo da cercare di massimizzare le ricompense future.

1.2 Reinforcement Learning

In questa sezione esaminiamo in modo più approfondito il reinforcement learning, perché DQN (l'algoritmo esaminato e modificato nel corso di questa tesi) rientra proprio in questa categoria.

Reinforcement learning (RL), grazie alla sua generalità, viene studiato in molte altre discipline, come game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms.

Gli elementi che lo rendono utile sono la capacità di apprendere da esempi, e l'utilizzo di un'approssimazione della funzione per la scelta delle azioni.

Questo permette di applicare algoritmi di RL anche in casi in cui non esiste una soluzione analitica esatta o l'unico modo di ottenere informazioni sull'ambiente è tramite l'interazione con esso.

Tipicamente, nei problemi di RL, l'ambiente di cui si parla è modellabile come un FMDP (Finite Markov Decision Process). Un FMDP è una tupla di quattro elementi (S, A, P_a, R_a), dove:

1. S rappresenta un insieme finito di stati,
2. A denota un insieme finito di azioni,
3. $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ esprime la probabilità che l'azione a nello stato s al tempo t conduca allo stato s' al tempo $t + 1$,
4. $R_a(s, s')$ indica la ricompensa ricevuta a seguito del passaggio dallo stato s allo stato s', tramite l'azione a

Questa è la descrizione generica di un FMDP, ricordiamo per inciso che nella maggior parte dei giochi Atari (l'ambiente della presente tesi sperimentale) le transizioni di stato sono deterministiche, questo significa che dato uno stato s ed un'azione a, $P_a(s, s')$ varrà 0 per tutti gli $s' \in S$ ad eccezione di un s^*, in cui ha valore 1.

$$P_a(s, s') = \begin{cases} 0 & \forall s' \in S, s' \neq s^* \\ 1 & \text{if } s' = s^* \end{cases} \tag{1.1}$$

Ad ogni time step t, l'agente riceve un'osservazione o_t, tipicamente equivalente allo stato s_t, e una reward r_t. In seguito l'agente compie un'azione a_t, che modifica l'ambiente dallo stato precedente s_t a s_{t+1}. Inoltre alla transizione (s_t, a_t, s_{t+1}) è associata una nuova ricompensa r_{t+1}. Questo ciclo continua fino al termine dell'episodio e lo scopo dell'agente è massimizzare la somma di tutte le ricompense ricevute (*cumulative reward*).

Ci sono casi in cui l'osservazione o_t dello stato s_t, contiene solo una parte di tutte le informazioni immagazzinate in s_t (es. per ambienti parzialmente osservabili). In altri casi, per poter trasmettere all'agente le informazioni

necessarie tramite stati e osservazioni, occorre comporre in un'unica osservazione più stati successivi. Ad esempio nel caso in cui occorrano informazioni su velocità e accelerazione, un'osservazione composta da un solo frame non sarebbe sufficiente.

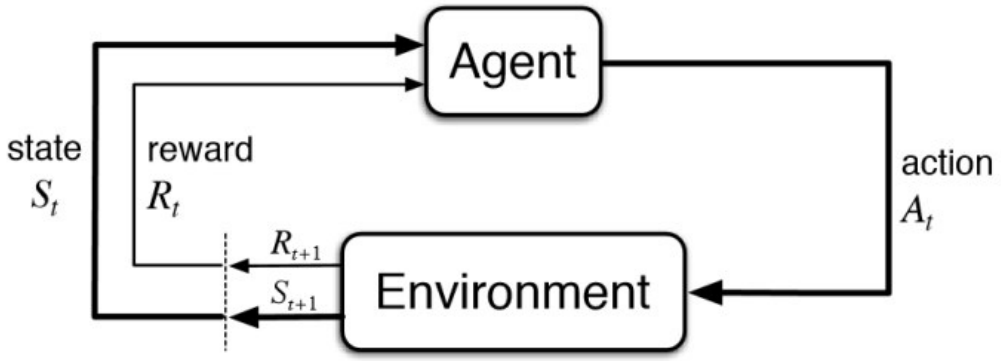

Figura 1.1: Schema di funzionamento astratto di un generico problema di RL.

Un fattore chiave in RL è l'esplorazione: abbiamo visto che l'algoritmo ha l'obiettivo di massimizzare il *cumulative reward* ma tipicamente, soprattutto per ambienti complessi, le strategie che apprende durante il corso dell'addestramento non sono perfette. È importante che l'agente non faccia costantemente la mossa che considera ottimale, ma continui anche a provare anche altre azioni e strategie perché potrebbero rivelarsi migliori.

Una strategia comunemente impiegata per mediare tra mossa migliore (exploitation) e esplorazione è la ε-greedy policy, che verrà illustrata nel prossimo capitolo, in quanto parte costitutiva dell'algoritmo DQN.

Capitolo 2

DQN Originale

Il motivo per cui nella redazione del presente progetto è stato utilizzato DQN (Deep Q-Network) è che la tesi prende le mosse dalla mia attività di tirocinio, il cui obiettivo era quello di modificare tale algoritmo. Quest'ultimo, come verrà illustrato in seguito più nel dettaglio, è nato da una modifica di Q-Learning che ne ha permesso l'estensione a problemi notevolmente più complessi non affrontabili con un approccio tabulare. DQN, a sua volta, presenta numerose versioni successive, sviluppate nel corso degli anni, e tuttora si continua a produrne.

Il focus preminente del lavoro svolto è stato quello di individuare ed implementare una serie di modifiche alla versione originale di DQN, con lo scopo di ottenere risultati migliori nel caso di giochi con ricompense sparse. Tali variazioni, che saranno esaminate separatamente nei prossimi paragrafi, sono state testate sia singolarmente sia in combinazione sui giochi Atari e confrontate con le performance di DQN originale.

2.1 Q-Learning

Q-learning (dove Q sta ad indicare *quality*) è un algoritmo *model-free* di *reinforcement learning*. Ciò significa che non si basa sull'uso delle funzioni di reward e di transizione di stato, ma si addestra a predire direttamente i

Q-values delle mosse dato lo stato corrente. Un Q-value è un valore numerico che l'algoritmo associa ad ogni possibile mossa e che rappresenta una stima della bontà della stessa. Un'azione risulta tanto più efficace quanto più è alto il Q-value; occorre precisare che nella stima dell'utilità di una mossa, Q-learning non considera solamente la reward immediata, ma cerca di tenere conto delle ricompense future e dunque del punteggio finale.

L'algoritmo inoltre è in grado di ottimizzare lo sfruttamento delle esperienze per l'addestramento: poiché si tratta di un metodo off-policy ciascun dato può essere utilizzato più volte. Q-learning si serve infatti di experience replay, un meccanismo che consente di salvare le esperienze estratte dalle simulazioni immagazzinandole in una memoria (detta replay buffer), da cui si estrapolano campioni casuali di dati che vengono utilizzati nell'addestramento al posto dell'ultima azione eseguita. Conseguenza diretta dell'impiego di un algoritmo off-policy è la diminuzione della correlazione tra i dati utilizzati nel training. Infatti esperienze provenienti da uno stesso episodio presentano un certo grado di interdipendenza; per contro, informazioni scelte in modo casuale dal buffer rivelano una correlazione nettamente inferiore e la policy risultante mostra maggiore stabilità durante l'addestramento.

È dimostrato che Q-learning individua la policy ottima nel caso di problemi riconducibili a Processi Decisionali di Markov Finiti. Questa è una proprietà rilevante dal momento che la maggior parte dei problemi di reinforcement learning (compresi i giochi Atari) possono essere modellizzati come FMDP.

2.2 DQN

DQN è stato il primo algoritmo di reinforcement learning ad utilizzare come input direttamente le immagini dell'ambiente, invece di servirsi di elaborazioni intermedie che estraggono indicatori numerici rilevanti [21]. Quest'ultima tecnica, che è stata utilizzata sino all'avvento di DQN, presentava i seguenti svantaggi:

- non è la rete neurale a dover capire quali siano le caratteristiche rilevanti ai fini del gioco, in quanto tali informazioni sono già implicite nei dati che le vengono forniti

- inoltre la rete dispone esclusivamente di quegli elementi che riceve e non di un quadro completo della situazione: ciò rappresenta evidentemente una limitazione, poiché tali valori non racchiudono necessariamente l'intera conoscenza disponibile e/o rilevante

- infine va sottolineato che con l'uso diretto dell'immagine (nuova tecnica) la struttura della rete diventa quasi totalmente indipendente dal problema esaminato; dato che l'input è sempre un'immagine di dimensione standardizzata, l'unica parte variabile diventa il numero di output della rete (pari al massimo numero di mosse possibili in uno stato). [1]

DQN rappresenta una risposta efficace a questi svantaggi in quanto l'input fornito alla rete è costituito da un'immagine che contiene tutte le informazioni disponibili; inoltre la forma dell'input non dipende dal gioco considerato e non ha quindi un impatto sulla struttura della rete.

Tali miglioramenti risulterebbero però vani se DQN non mostrasse performance positive durante i test. Alla prova dei fatti l'algoritmo ha manifestato una competenza paragonabile a quella di esperti umani e in molti casi addirittura superiore nell'espletamento di giochi Atari 2600 [20] (disponibili su ALE, Arcade Learning Environment [3]).

[1] In DQN l'unica parte della struttura della rete dipendente dall'ambiente è la quantità degli output (valore del Q-value di ogni azione in quello stato). È possibile utilizzare una rete neurale che prende in input la coppia stato-azione e restituisce un unico output che è il Q-value di tale azione in quello stato. In questo caso anche il numero di output è indipendente dall'ambiente e fisso, ma non si utilizza questa architettura perché richiede una quantità di calcoli molto maggiore. Il vantaggio di usare la prima architettura è che con un solo forward pass (interrogare una volta la rete neurale) si ottengono i Q-value di tutte le mosse, mentre nel secondo caso avere queste informazioni ha un costo lineare sul numero di mosse totali.

La logica di funzionamento di DQN a livello astratto è la seguente: scelto un ambiente e un relativo stato, una rete neurale viene addestrata ad assegnare un valore (Q-value) ad ogni possibile azione; con queste informazioni è poi possibile scegliere l'azione migliore. Operativamente l'algoritmo simula tantissimi episodi accumulando le mosse fatte ed altri dati in un buffer di "esperienze"; nel frattempo progressivamente e casualmente seleziona esperienze dal suddetto buffer per addestrare la DNN. Le esperienze sono costituite da tuple (s_t, a_t, r_t, s_{t+1}) e contengono quindi le seguenti informazioni:

- s_t, stato del sistema al tempo t
- a_t, azione eseguita nello stato s al tempo t
- r_t, reward ricevuta a seguito di a_t
- s_{t+1}, stato del sistema al tempo $t+1$, a seguito di a_t

Per ragioni di efficienza, nell'implementazione di DQN utilizzata, tali tuple contengono il valore aggiuntivo "done": $(s_t, a_t, r_t, s_{t+1}, done)$. "done" è una variabile booleana che indica se lo stato s_t è terminale. Uno stato si dice terminale quando è l'ultimo stato di un episodio, indipendentemente dalla causa: la "morte" dell'agente o il raggiungimento del numero massimo di step.

Il modo in cui la rete impara dalle esperienze è basato sull'equazione di Bellman:

$$Q^*(s,a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \tag{2.1}$$

Si tratta di un processo iterativo in cui la rete vede come errore da minimizzare la differenza fra i due termini dell'equazione.

γ (*gamma discount factor*) rappresenta il peso che vogliamo attribuire alle ricompense future rispetto a quelle immediate: con $\gamma = 0$ vengono considerate solo le ricompense immediate, con $\gamma = 1$ la rete non predilige in alcun modo le ricompense immediate a scapito di quelle future. Un valore tipico, usato anche in questo progetto, è $\gamma = 0.99$, che dà molta importanza alle

ricompense future, ma predilige leggermente quelle più immediate. Questa scelta ha due vantaggi importanti:

1. permette la convergenza anche in caso di problemi con numero di mosse potenzialmente illimitato

2. fintanto che la rete non impara a giocare in modo perfetto (tipicamente non succede nemmeno alla fine dell'addestramento), le stime delle ricompense future più lontane nel tempo diventano via via meno accurate, quindi attribuire un maggior peso a quelle più precise (le più immediate) produce un comportamento migliore.

La conoscenza della variabile "done" riferita alle esperienze risulta rilevante poiché gli stati terminali permettono di migliorare le stime ottenute dall'equazione di Bellman: se infatti $\bar{s}$ è uno stato terminale, il calcolo del corrispondente Q-value si riduce a:

$$Q^*(\bar{s}, a) = r \tag{2.2}$$

Nel caso degli stati terminali conosciamo infatti con certezza, per ogni azione, il punteggio massimo (cioè l'unico) raggiungibile, perché non esistono successivi stati o azioni future da considerare.

Alcuni accorgimenti importanti utilizzati in DQN sono clipping delle reward, preprocessing delle immagini, frame-skipping, target network, experience replay e ε-greedy policy descritti di seguito in modo più dettagliato.

Clipping delle reward

Si introduce per uniformare giochi che usano scale di punteggio molto differenti in modo da non dover cambiare iperparametri (come il learning rate) tra l'uno e l'altro. Questa tecnica consiste nel limitare le reward positive e negative a valori prestabiliti (in questo caso rispettivamente $+1$ e -1), mentre le reward nulle rimangono invariate.

Preprocessing delle immagini

Anche quando le immagini sono usate direttamente come input, è comunque necessario eseguire alcune operazioni di preprocessing su di esse prima di passarle alla rete, ad esempio nei giochi Atari:

- alcuni oggetti compaiono solo in frame dispari e altri in frame pari (Atari 2600 riusciva a mostrare solo un certo numero massimo di oggetti alla volta). In questo caso il preprocessing consiste nel riprodurre tutti gli oggetti in ogni frame
- si rendono i frame in bianco e nero
- si riscalano a dimensioni 84×84 (da 210×160)

Frame-skipping

Il costo computazionale che ha l'agente per scegliere una mossa è molto maggiore di quello dell'emulatore per simulare uno step di gioco. La tecnica di frame-skipping consiste nel far scegliere una mossa all'agente solo ogni k frame, e questa mossa viene mantenuta per tutti i k frame. In questo modo si ottiene un numero di esperienze k volte superiore rispetto a DQN senza frame-skipping, per un aumento di tempi di simulazione molto piccolo (tipicamente si usa $k = 4$).

Target network

Un problema che può causare instabilità nell'addestramento e impedire la convergenza verso una strategia ottima è l'interdipendenza fra gli action values (Q) e i target values ($r + \gamma \max_{a'} Q(s', a')$). In particolare mentre la rete modifica i propri pesi per ottenere action values sempre più simili ai target values, la modifica agli action values causa una modifica nell'obiettivo, i target values. Per ovviare a questo problema i target values vengono presi dal target network. Il target network è una seconda rete neurale identica al Q-network principale, con la differenza che essa non viene addestrata. Invece ogni C step il target network copia i pesi dal Q-network, in questo modo i target values rimangono molto più stabili.

Experience replay

Un'altra caratteristica chiave di DQN è experience replay: ogni esperienza generata dall'agente invece di essere usata direttamente per addestrare la rete e poi scartata, viene semplicemente immagazzinata in un buffer (replay buffer). Dopo ogni 4 azioni eseguite dall'agente (perché "train_freq" = 4), la rete si addestra su un'intero batch di esperienze estratte casualmente da questo buffer. Questa tecnica comporta tre vantaggi principali:

1. data efficency; ogni esperienza, una volta salvata nel replay buffer, può essere estratta ed utilizzata più volte per addestrare la rete

2. riduce la correlazione tra le esperienze di training, quindi riduce la varianza degli aggiornamenti

3. se si addestrasse direttamente in ogni step con l'esperienza generata dalla mossa eseguita, l'algoritmo rischierebbe di bloccarsi in un massimo locale, perché la scelta delle mosse da effettuare (e quindi l'esplorazione) dipende dalla policy attuale. Con il replay buffer la rete si allena anche su mosse diverse da quelle che compierebbe attualmente.

ε-greedy policy

Finora non abbiamo parlato esplicitamente della policy π con cui l'algoritmo sceglie le mosse da effettuare. Dato uno stato del gioco, il Q-Network restituisce un Q-value per ogni possibile mossa, quindi sembrerebbe naturale scegliere sempre la mossa a cui è associato il Q-value maggiore:

$$\pi(s_t) = \arg\max_{a' \in \mathcal{A}} Q(s_t, a'). \tag{2.3}$$

In realtà in DQN si utilizza la ε-greedy policy, che consiste nello scegliere la mossa ritenuta migliore con probabilità $1 - \varepsilon$, e scegliere una mossa casuale con probabilità ε:

$$\pi'(s_t) = \begin{cases} \text{random action } a \in \mathcal{A} & \text{with probability } \varepsilon \\ \arg\max_{a' \in \mathcal{A}} Q(s_t, a') & \text{with probability } 1 - \varepsilon \end{cases} \tag{2.4}$$

Il motivo per cui si utilizza la ε-greedy policy è per avere un parametro (appunto ε) che permetta di controllare il rapporto exploration-exploitation,

inoltre per ridurre la possibilità che l'algoritmo rimanga intrappolato in massimi locali, lasciando una piccola probabilità di esplorazione anche quando DQN "crede" di aver trovato una strategia efficace.

2.3 Versioni preesistenti di DQN

L'algoritmo DQN è stato ulteriormente sviluppato ed ha rappresentato il punto di partenza per l'elaborazione di numerose versioni successive. Alcune di esse sono state incorporate nella versione contenente anche le modifiche apportate da me all'algoritmo, mentre altre sono state testate come avversarie.

Di seguito passeremo rapidamente in rassegna le principali evoluzioni di DQN corredate da una breve spiegazione [14]:

- Deep Recurrent Q-Learning (DRQN) è una variante che mostra risultati ottimali nel caso di giochi ad informazione incompleta (in cui gli agenti decisionali dispongono di informazioni parziali sullo stato del gioco e sulle scelte degli altri agenti); sostanzialmente amplia l'architettura di DQN con l'aggiunta di un layer ricorrente prima dell'output.

- Gorila DQN è una versione distribuita di DQN che, in 41 dei 49 giochi Atari utilizzati come test, ha ottenuto punteggi superiori rispetto alla controparte originale. Tale algoritmo è stato creato servendosi di Gorila (General Reinforcement Learning Architecture), che permette la parallelizzazione degli agenti durante il training, per una collezione di esperienze meno correlate ed un conseguente addestramento più stabile.

- Double DQN [30], che si fonda su double Q-learning, rappresenta una versione dell'algoritmo così largamente accettata e consolidata che, ad oggi, viene considerata parte integrante di DQN e compare in diverse modifiche successive. Double DQN nasce per porre rimedio ad un noto problema legato all'algoritmo originale, ovvero la sovrastima dei valori

delle mosse (Q-values). Per farlo si serve di due distinte funzioni di valutazione apprese da due reti neurali separate, rispettivamente l'online network con parametri θ e il target network con parametri θ^-. In seguito imposta il calcolo per individuare l'azione migliore scomponendolo in due diverse operazioni: valutazione della greedy policy (procedimento svolto facendo uso dei valori dell'online network), e stima dei valori (condotta utilizzando il target network). Il calcolo del target Y_t^{DQN} risulta dunque il seguente:

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}; \arg\max_a Q(S_{t+1}; a; \theta_t); \theta_t^-) \qquad (2.5)$$

- Prioritized Replay [28] sfrutta la tecnica del prioritized experience replay che consente di estrarre esperienze dal replay buffer attribuendo una priorità maggiore a quelle più significative. La scelta delle esperienze più importanti si fonda sul TD error, prediligendo quindi informazioni con un maggior valore di apprendimento atteso.

- Dueling DQN [32] è una versione in cui la rete neurale, superati gli strati convoluzionali, si ramifica in due flussi per realizzare due distinte operazioni: uno di essi svolge la funzione di calcolare lo state-value $V^\pi(s)$, l'altro compie una stima dell'action-advantage $A^\pi(s; a)$. Combinando i due valori appena menzionati si ottiene il Q-value desiderato: $Q^\pi(s; a) = V^\pi(s) + A^\pi(s; a)$. Il vantaggio dell'operazione appena esposta è il seguente: i due valori vengono calcolati in maniera accurata e di conseguenza anche la stima del Q-value che ne deriva risulta migliore rispetto a quella calcolata in DQN originale.

- Bootstrapped DQN dispone di numerose reti neurali e, durante ogni episodio, ne seleziona una in modo casuale per completare la partita ed essere quindi addestrata. Tale procedimento garantisce un'esplorazione migliore dal momento che le varie reti ricevono training differenti, e questo le porta a strategie dissimili. L'impiego delle cosiddette bootstrap masks, che alterano i gradienti delle varie reti neurali portan-

do ad una conseguente variazione delle policy implementate, assicura un'esplorazione maggiormente ampia e diversificata.

- Distributional DQN [4] si differenzia dall'algoritmo di base poiché ciascun Q-value restituito dalla funzione $Q(s;a)$ non risulta costituito da un singolo valore, ma si configura come una distribuzione discreta di probabilità. I valori che la distribuzione può assumere sono detti atomi; a parità di dimensione dell'intervallo, all'aumentare del numero di atomi e quindi della granularità, si osserva un miglioramento dei risultati.

- NoisyNet-DQN consiste in una versione il cui metodo di esplorazione si contrappone a quello standard della ε-greedy policy: ad ogni parametro della rete viene applicata una specifica fluttuazione detta noise. Essa introduce un fattore di casualità nella scelta delle mosse rispetto a quelle che verrebbero altrimenti eseguite attenendosi alla policy di base. Il valore delle fluttuazioni, che all'inizio è casuale, viene progressivamente affinato tramite la discesa del gradiente.

- Rainbow si configura come una combinazione di alcune delle modifiche di cui sopra; in particolare utilizza: Double DQN, Prioritized Replay, Dueling DQN, Distributional DQN, e NoisyNets. I dati mostrano che in media ottiene punteggi superiori rispetto alle singole versioni considerate individualmente.

- Parameter Space Noise [25] (utilizzabile anche per altri algoritmi oltre a DQN) cerca di migliorare l'esplorazione effettuata dagli algoritmi, applicando piccole perturbazioni all'inizio di ogni episodio direttamente ai pesi della rete neurale adibita all'esplorazione. In questo modo, a differenza di scelte casuali "forzate" esternamente, la rete ha un comportamento più consistente all'interno della stessa partita (ad es. se una situazione si presenta più volte nel corso di un episodio, normalmente l'algoritmo può fare ogni volta mosse diverse, mentre con Parameter Space Noise tenderà a mantenere la stessa azione).

Capitolo 3

Modifiche a DQN

In questo capitolo verranno esposte le modifiche da me apportate a DQN ai fini della presente tesi. Alcune di esse sono concepite come estensioni volte a migliorare la performance dell'algoritmo nel complesso, mentre altre sono tese alla risoluzione di problematiche più specifiche. Trattandosi di variazioni sperimentali, l'esito si è rivelato piuttosto eterogeneo: certe modifiche hanno registrato effetti pressoché nulli, altre hanno permesso di conseguire risultati soddisfacenti soltanto in alcuni dei giochi Atari ed altre ancora hanno mostrato miglioramenti significativi soltanto in una fase iniziale dell'addestramento. Di seguito forniremo una presentazione approfondita di tali estensioni.

3.1 Lower Bound DQN

La prima versione, denominata Lower Bound DQN, si fonda sull'aggiunta di un buffer ulteriore rispetto a quello originale, col fine di immagazzinare dati relativi al valore minimo (lower bound) che il Q-value di una data mossa in un determinato stato del gioco può assumere. A differenza dei Q-value, il lower bound non costituisce dunque una stima ma il risultato di un calcolo esatto, dal momento che alla fine di ogni episodio si dispone di tutte le informazioni necessarie per calcolarlo.

La natura degli elementi contenuti nel buffer aggiuntivo è molto simile a quella delle esperienze memorizzate nel replay buffer, ma ogni lower bound viene trattato come uno stato terminale.

L'obiettivo finale dell'estensione è quello di consentire all'algoritmo l'ottenimento di risultati migliori nel caso di giochi con ricompense sparse.

Vale la pena introdurre una precisazione: un addestramento di DQN si compone di numerose partite (o episodi), a loro volta suddivise in una sequenza di mosse. Occorre segnalare il diverso svolgimento delle fasi di memorizzazione ed estrazione rispettivamente nel caso del replay buffer e del lower bound (LB) buffer. Per ogni mossa, la relativa esperienza viene immagazzinata direttamente nel replay buffer, mentre il lower bound viene calcolato soltanto al termine della partita, per essere poi memorizzato nel LB buffer. Il momento dell'estrazione presenta invece un maggiore grado di somiglianza: ad intervalli di 4 mosse vengono estratte 16 esperienze e 16 lower bound utilizzati per addestrare la rete (diversamente da quanto avviene utilizzando DQN originale, dove si estrapolano 32 esperienze).

Di seguito viene brevemente presentata ed illustrata la formula relativa al calcolo dei lower bound:

$$LB(s_t, a_t) = \sum_{i=t}^{T} r_i \times \gamma^i \tag{3.1}$$

Il lower bound ottenuto per ogni mossa al termine della partita viene ricavato sommando al punteggio della mossa in questione quello di tutte le mosse successive, con γ. Il fattore di sconto γ viene introdotto nel calcolo dei lower bound per far sì che l'importanza di una ricompensa attesa diminuisca all'aumentare della sua lontananza nel tempo (espressa in numero di mosse necessarie per ottenerla).

L'equazione appena esposta rappresenta il motivo per cui i lower bound vengono considerati stati terminali. La reward r_t memorizzata nella tupla non si riferisce soltanto alla mossa appena effettuata ma contiene tutte le ricompense future fino al termine dell'episodio, dunque costituisce una cu-

mulative reward (ciò è possibile perché i lower bound vengono calcolati solo al termine di ogni episodio).

Prima di poter eseguire addestramenti e test è stato necessario stabilire la policy che disciplina l'estrazione di batch di esperienze dal replay buffer e di lower bound dal lower bound buffer, nonché la loro memorizzazione nei rispettivi buffer (come accennato sopra).

Le policy sperimentate sono state due. La prima prevede di salvare tutti i lower bound calcolati a conclusione di una partita nel LB buffer e, a seguito dell'estrazione, contempla l'esecuzione di un test per ognuno dei 16 lower bound. Occorre sottolineare che l'estrazione rappresenta solamente un'operazione di copia di elementi dal buffer e non comporta una loro effettiva rimozione dallo stesso. Il test consiste nell'interrogare la rete sul valore del Q-value associato allo stato e all'azione relativi a ciascuno dei 16 lower bound. Se il Q-value stimato dalla rete risulta superiore al valore del lower bound in questione, quest'ultimo verrà scartato poiché l'informazione contenuta nel lower bound non è più rilevante. Si può infatti dire che il lower bound migliora l'addestramento di una rete neurale in quelle circostanze in cui essa assocerebbe un Q-value eccessivamente basso a determinate coppie stato-azione. Va da sé che se il Q-value calcolato dalla rete risulta superiore rispetto al lower bound, che per definizione dovrebbe consistere nel suo valore minimo, allora il lower bound si rivela superfluo. In conclusione, dei 16 LB estratti, vengono impiegati nell'addestramento della rete soltanto quelli che superano il test. I restanti, comunque, non vengono eliminati dal buffer e possono quindi comparire in estrazioni successive. Per far sì che la dimensione del batch di addestramento rimanga invariata (32 elementi), si compensano gli eventuali lower bound scartati con l'aggiunta di altrettante esperienze nel replay buffer, che si sommano alle 16 originali.

La seconda policy si differenzia da quella appena enunciata per le seguenti peculiarità: al termine di ogni partita non vengono indiscriminatamente salvati tutti i lower bound ricavati; infatti, il test che nella policy precedente si effettua successivamente all'estrazione dei batch di addestramento, viene qui

eseguito prima della memorizzazione dei lower bound nel rispettivo buffer. Dunque, in questo caso, i LB che non superano il test non vengono immagazzinati nel buffer. Di conseguenza, i batch di addestramento risultano già composti di 16 unità ciascuno e non richiedono ulteriori modifiche.

Dopo svariate prove, si è deciso di utilizzare la seconda policy. Dalle simulazioni eseguite è infatti emerso che, nel primo caso, il lower bound buffer riceve, durante l'addestramento, una quantità crescente di elementi superflui, che possono essere di volta in volta estratti al pari di quegli LB che risultano per contro utili ai fini del training. Con la seconda policy si ovvia a tale problematica, poiché la selezione degli elementi avviene a monte e dunque il buffer finisce per contenere, in proporzione, un numero molto maggiore di LB rilevanti.

Per concludere l'esposizione della versione Lower Bound DQN, occorre segnalare che è stato introdotto un ulteriore accorgimento rispetto all'algoritmo originale: nella fase di memorizzazione si è infatti stabilito che i lower bound vengono inseriti nel relativo buffer unicamente per le partite che si concludono con punteggi positivi. Si è scelto di integrare tale modifica nella versione corrente perché esiste una convergenza di obiettivi con LB DQN: entrambe hanno di fatto lo scopo di migliorare la performance dell'algoritmo nel caso di giochi a ricompense sparse.

3.2 Massima esplorazione fino alla prima ricompensa

Nella prima fase, con ε-greedy policy, ε parte da 1 (mossa totalmente casuale) e si riduce progressivamente fino a raggiungere il minimo, deciso inizialmente tramite un parametro (di default 0.01). Utilizzando un secondo parametro si controlla quanto è grande la frazione di simulazione in cui ε cala, nella seconda fase ε rimane fisso al valore minimo.

La modifica esposta[1] in questo capitolo consiste nell'aggiungere un'altra fase, precedente le prime due, in cui ε rimane fisso ad 1 (valore massimo) e viene quindi eseguita sempre una mossa casuale; l'algoritmo può passare alla fase successiva solo dopo aver ottenuto la prima ricompensa positiva.

L'idea alla base di questa modifica è rappresentata dal fatto che finché DQN non riceve informazioni su quali siano le mosse migliori (sotto forma di reward positive), la strategia appresa dall'algoritmo non può essere una buona strategia; quindi la gestione ottimale consiste nel mantenere esplorazione massima (cioè mossa totalmente casuale).

3.3 Salvataggio e Caricamento di un addestramento interrotto

Il codice fornito in OpenAI Baselines fornisce già funzioni per salvare e caricare un modello (cioè i pesi della rete neurale), ma queste sono progettate per uno scopo preciso e quindi limitate ad esso. Dopo aver totalmente concluso un addestramento è possibile utilizzare la funzione di salvataggio per memorizzare i pesi; successivamente, se si vogliono vedere i risultati di quel particolare addestramento, basta caricare i pesi precedentemente memorizzati e si può vedere l'algoritmo in azione.

Tutto questo non è un problema in sé, il problema sorge quando si voglia o si debba riprendere un'addestramento interrotto, non è infatti possibile questa operazione utilizzando solo gli elementi memorizzati dalla funzione di salvataggio. Es. si vuole spezzare l'addestramento in due, svolgere solo la prima parte, salvare e interrompere. Successivamente si vogliono caricare di nuovo i pesi della rete parzialmente addestrata ed eseguire la seconda parte dell'addestramento. Con queste funzioni non è possibile compiere queste operazioni perché oltre ai pesi della rete servono molti altri dati che non vengono salvati.

[1] Per i giochi a punteggi negativi questa modifica non ha effetto. In alcuni casi non erano chiare le motivazioni dei punteggi negativi, inoltre si tratta di una casistica ridotta.

Ho quindi realizzato una funzione di salvataggio e la corrispondente funzione di caricamento che raccolgono più informazioni, come di seguito elencato:

- i pesi attuali della rete
- i parametri inseriti all'avvio dell'algoritmo
- il contenuto del replay buffer
- il contenuto del lower bound buffer
- altri dati specifici calcolati all'inizio dell'algoritmo

Per l'effettivo salvataggio dei dati in un unico file uso cloudpickle [7], una estensione di pickle pensata proprio per essere eseguita su host remoti. In questo modo il file viene creato sulla macchina virtuale fornita da colab e appena questa viene chiusa tutti i file che contiene vengono eliminati. Quindi, per ovviare a questo inconveniente, ho scritto una parte di codice esterna al programma, direttamente nel foglio colab, che al termine dell'esecuzione delle simulazioni, si connette a Google Drive e inserisce il risultato dell'addestramento (cioè il file di salvataggio di cui parlavamo sopra) in una cartella apposita. In questo modo una copia del file che contiene tutte le informazioni rilevanti è salvata in modo permanente e indipendente dalla macchina virtuale.

È importante salvare anche i buffer perché si evita nella ripresa dell'esecuzione di eseguire step "a vuoto" come serve fare all'inizio dell'addestramento, con il solo scopo appunto di riempire i buffer. Inoltre quando si carica una sessione predentemente interrotta, la variabile ε della ε-greedy policy ripartirebbe da 1, ma in questo caso è inefficace che la rete esplori facendo mosse casuali dal momento che è già addestrata almeno parzialmente; quando si carica, quindi, ε è direttamente fissato al valore finale.

Le nuove funzioni che ho implementato sono quindi in grado di ottenere gli stessi risultati di quelle fornite da OpenAI Baselines. In più permettono di

riprendere l'addestramento interrotto risolvendo la mancanza delle funzioni originali. Si è però visto dai test che è presente un nuovo problema: per un certo numero di step di addestramento seguenti al caricamento di una sessione precedente, la rete ha una performance di gioco particolarmente bassa (come se fosse non addestrata), poi risale velocemente fino alla performance precedente, e poi a questo punto riprende ad addestrarsi normalmente. Dopo vari tentativi di analisi non sono comunque riuscito a risalire alla causa del problema (potrebbero esserci altri dati rilevanti, come lo stato dell'ottimizzatore, non presi in considerazione). Per questo le simulazioni sono state condotte rispettando i tempi massimi di elaborazione resi disponibili da colab senza segmentare l'addestramento in più fasi.

3.4 Stochastic DQN

La ε-greedy policy ha alcuni aspetti negativi:

- in un caso l'algoritmo esegue la mossa che considera migliore ignorando il resto delle mosse, tra cui anche altre mosse "buone"
- nell'altro caso l'algoritmo fa una mossa totalmente casuale, ignorando tutto quello che ha imparato sui valori delle mosse in quella situazione

Si è ritenuto quindi conveniente testare una policy più organica, che cerchi di scegliere le mosse in modo stocastico, ma con probabilità collegate ai Q-value.

Es. Si suppongano tre possibili mosse con Q-value 5, 4, 0.1. Con ε-greedy policy verrebbe scelta quasi sempre la prima mossa, anche se la seconda è quasi egualmente utile, e pochissime volte verrebbe eseguita una scelta con probabilità uniforme tra le tre azioni, senza pesare il fatto che la terza mossa è pessima rispetto alle altre due quasi egualmente valide. Si vorrebbe invece realizzare un metodo che in un caso come questo scelga casualmente spesso la prima mossa e abbastanza spesso anche la seconda, ma pochissime volte l'ultima.

Sono state implementate e provate alcune varianti a partire da questa idea.

1. La scelta delle mosse è direttamente proporzionale al valore dei rispettivi Q-value. Per ottenere le probabilità si calcola un *softmax*, in modo da normalizzare a 1 la somma delle probabilità risultanti.

   Questa strategia introduce nuovi problemi:

   (a) o si combina comunque con ε-greedy oppure non si ha esplorazione decrescente (non c'è un parametro regolabile per prediligere esplorazione o punteggio)

   (b) come viene spiegato nel paper su Dueling DQN [32], i Q-value dipendono in gran parte dalla bontà o meno dello stato (che è uguale per tutte le mosse) e solo in piccola parte dalla bontà delle singole mosse. Quindi i Q-value spesso sono tutti molto simili tra loro, di conseguenza le probabilità risultanti sono anch'esse molto simili e questo comporta una scelta quasi casuale delle mosse

2. Per risolvere il problema (b), si è pensato di sostituire i Q-value, con il rank delle mosse. Cioè l'azione con Q-value più alto ha sempre una probabilità certa di essere scelta indipendentemente da quanto è più alto il suo Q-value rispetto a quello delle altre mosse, stessa cosa per la seconda mossa e per tutte le mosse successive.

   Resta comunque il problema (a), non si ha un modo per prediligere fra exploration e exploitation.

3. Per risolvere il problema (a), si è provato ad incorporare ε nella prima strategia, in modo da avere un parametro che controlli il grado di esplorazione. In questo caso, a differenza di ε-greedy policy, ε parte da 0 e cresce fino ad un numero grande (massimo 100 perché con valori maggiori si genera overflow in un risultato parziale). Tecnicamente l'operazione consiste nell'elevare i Q-value ad ε prima della normalizzazione con softmax.

All'inizio ($\varepsilon = 0$) si ha il classico comportamento esplorativo dato che tutti i Q-value elevati a 0 risultano 1 (quindi le probabilità sono tutte uguali fra loro, e la scelta è uniforme). Finchè $0 < \varepsilon < 1$, le differenze fra i Q-value risultano ridotte nelle probabilità finali calcolate in questo modo, mentre dal momento in cui $\varepsilon > 1$ le piccole differenze fra i Q-value vengono accentuate sempre di più nelle probabilità finali.

Questa tecnica risolve anche il problema (b) ma solo parzialmente, dato che esiste un limite al valore massimo di ε (100). Si è infatti osservato sperimentalmente che i risultati migliorano all'aumentare del valore massimo di ε.

4. Combinando la seconda e la terza strategia si arriva alla versione migliore tra quelle sperimentate: si usano i rank delle mosse determinati dai Q-value invece di usare direttamente i Q-value, si elevano ad ε ed infine si normalizzano con *softmax*.

   In questo modo con il parametro ε possiamo controllare l'esplorazione (maggiore è ε minore è l'esplorazione) e con l'utilizzo del rank si risolve il problema di Q-value troppo simili fra loro.

3.5 Entropy DQN

L'ultima modifica presentata è un adattamento di una delle strategie utilizzate da Acer [33] e A3C [19] (altri due algoritmi di Reinforcement Learning). Nel caso specifico di Montezuma's Revenge (uno dei giochi Atari, noto per la sua altissima difficoltà), Acer sembrava riuscire ad ottenere risultati superiori a DQN, in particolare per quanto riguarda l'esplorazione senza o con pochissime informazioni/ricompense. L'ipotesi formulata è che tra le varie differenze fra Acer e DQN, quella rilevante in questo caso sia rappresentata dall'introduzione dell'entropia. Per il codice si è fatto sempre riferimento ad OpenAI Baselines.

Le motivazioni principali dell'inserimento dell'entropia sono una migliore esplorazione in condizioni di scarse informazioni [2] e un overfitting prematuro verso un massimo locale.

L'entropia modifica le probabilità di scelta delle mosse cercando di contenere le ripetizioni di una stessa mossa o di intere sequenze di mosse.

L'entropia, per essere calcolata, richiede di conoscere le probabilità relative alla scelta delle azioni, e grazie alla modifica precedentemente illustrata (sezione 3.4), sono proprio disponibili queste informazioni, dato che ho introdotto in DQN la scelta delle mosse in modo stocastico.

Di seguito la formula utilizzata per il calcolo dell'entropia:

$$H = -\sum_i P_i \log P_i \tag{3.2}$$

dove H indica l'entropia e P rappresenta una distribuzione di probabilità. Nel nostro caso specifico P è la distribuzione di probabilità sulla scelta delle azioni, cioè $P = \pi$ e P_i indica la probabilità di scegliere la i-esima mossa (π indica come di consueto la policy di scelta delle mosse).

Sono state realizzate due versioni di Entropy DQN:

1. la prima, "parziale", funziona come DQN originale (mosse scelte in modo deterministico a parte ε-greedy policy); le probabilità associate alle mosse vengono calcolate come in Stochastic DQN ma vengono utilizzate esclusivamente per il calcolo dell'entropia e non per la scelta delle azioni da compiere.

2. la seconda, "completa", si avvale delle probabilità associate ad ogni azione sia per il calcolo dell'entropia sia per la scelta delle mosse.

La seconda variante è quella concettualmente più corretta, in quanto nel primo caso le probabilità associate alle mosse non corrispondono alla vera frequenza con cui vengono effettivamente selezionate le azioni, ma ho comunque deciso di testare entrambe per valutarne comparativamente i risultati.

"entropy_coeff" è un parametro che viene utilizzato per scalare l'entropia a valori con un ordine di grandezza paragonabile a quello dei Q-value, nel caso di questo progetto il valore scelto è lo stesso del paper su A3C, cioè 0.01.

3.6 Note su iperparametri e modifiche ad un loro sottoinsieme

Nelle tabelle seguenti vengono elencati gli iperparametri utilizzati con i relativi valori di default, le righe in grassetto indicano particolari iperparametri di cui sono stati testati anche altri valori. La prima tabella (Tabella 3.1 "DQN") raccoglie tutti gli iperparametri utilizzati in ogni versione implementata; le tabelle successive contengono solo iperparametri specifici o modifiche a quelli della prima tabella per versioni specifiche di DQN. Ad esempio la tabella "Lower Bound DQN" indica che nella versione di DQN con lower bound, "batch_size" viene sostituito da due iperparametri: "replay_batch_size" e "lb_batch_size".

Serve precisare inoltre che certe variabili non vengono riportate in quanto non hanno rilevanza sul risultato delle simulazioni, ad esempio "print_freq", la frequenza con cui vengono stampate informazioni in output durante il corso della simulazione.

Per ognuna delle tabelle forniamo una breve descrizione degli iperparametri elencati, per poi esaminare più in specifico quelli su cui sono stati eseguiti test con diversi valori.

- Tabella 3.1 "DQN"
  - "lr": learning rate dell'ottimizzatore, in questo caso Adam Optimizer. Il learning rate controlla quanto influiscono nuove esperienze sugli attuali pesi della rete.
  - "batch_size": dimensione del gruppo di esperienze da estrarre dal replay buffer e su cui addestrare la rete dopo ogni azione eseguita in un episodio.
  - "target_network_update_freq": frequenza (espressa in numero di mosse) con cui aggiornare il target network (vedi target network).
  - "gamma": il gamma discount factor rappresenta il peso delle ricompense future rispetto a quelle immediate. Stabilisce un com-

DQN	
lr	10^{-4}
batch_size	32
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	$\mathbf{4 \times 10^6}$
exploration_fraction	**0.1**
exploration_final_eps	**0.01**

Tabella 3.1: Iperparametri di base utilizzati in tutte le versioni di DQN (salvo variazioni esplicite).

promesso fra mosse che portano ad una possibile ricompensa finale molto alta e mosse che con sicurezza danno punti velocemente (vedi gamma discount factor).

– "train_freq": frequenza (espressa in numero di mosse) con cui addestrare la rete neurale con le esperienze raccolte e memorizzate durante le partite.

– "buffer_size": dimensione del replay buffer, in cui vengono immagazzinate le esperienze (mosse con relativi risultati e conseguenze) effettuate durante gli episodi di training. Il buffer è aggiornato in maniera ciclica: ogni volta che una nuova esperienza deve essere inserita ma il buffer è già pieno, la più vecchia esperienza viene cancellata per fare spazio.

– "learning_starts": numero di mosse da effettuare inizialmente pri-

Lower Bound DQN			
batch_size	$\Longrightarrow$	replay_batch_size	32
		lb_batch_size	16

Tabella 3.2: Differenze fra gli iperparametri generali e quelli di Lower Bound DQN.

Stochastic DQN			
exploration_final_eps	0.01	$\Longrightarrow$	100

Tabella 3.3: Differenze fra gli iperparametri generali e quelli di Stochastic DQN.

ma di usarle per addestrare la rete. Tipicamente viene scelto uguale a "buffer_size", in modo che l'addestramento inizia esattamente quando il replay buffer è stato riempito.

- "param_noise": è un booleano che stabilisce se usare la versione di DQN introdotta in "Parameter Space Noise for Exploration" [25].
- "prioritized_replay": come sopra, si tratta di una variabile che controlla l'attivazione o meno delle modifiche esposte in "Prioritized experience replay" [28].
- "dueling": anche in questo caso questo parametro booleano permette di scegliere se usare DQN base o Dueling DQN [32].
- "total_timesteps": numero totale di azioni dopo le quali l'addestramento della rete neurale è concluso.
- "exploration_fraction": attraverso la ε-greedy policy, stabilisce la percentuale di mosse usate per l'esplorazione iniziale rispetto al totale delle mosse disponibili ("total_timesteps").
- "exploration_final_eps": limite inferiore al valore di ε (relativo alla ε-greedy policy). Quindi indica la probabilità che l'algoritmo

Entropy DQN	
entropy_coeff	0.01

Tabella 3.4: Differenze fra gli iperparametri generali e quelli di Entropy DQN.

esegua mosse casuali (cioè che provi ad testare nuove strategie), una volta che la fase di esplorazione iniziale è terminata.

- Tabella 3.2 "Lower Bound DQN"
  - "batch_size": viene sostituito da due iperparametri più specifici: "replay_batch_size" e "lb_batch_size". Questo cambiamento viene reso necessario dal fatto che in questa variante si hanno due buffer invece di uno solo (oltre al replay buffer è presente anche il lower bound buffer).
  - "replay_batch_size": mantiene circa la funzione del precedente "batch_size". La sua funzione è di stabilire la quantità di esperienze totali (indipendentemente dal buffer di provenienza) su cui la rete neurale su addestra ad ogni "train_freq" azioni.
  - "lb_batch_size": indica quante delle "replay_batch_size" esperienze vengono estratte dal lower bound buffer. Di conseguenza controlla anche quante ne vengono estratte dal replay buffer (cioè "replay_batch_size" − "lb_batch_size").
- Tabella 3.3 "Stochastic DQN"
  - "exploration_final_eps": come spiegato nella relativa sezione, in Stochastic DQN ε ha valore iniziale 0 e aumenta durante l'addestramento. In questo caso "exploration_final_eps" rappresenta quindi un limite superiore per ε (vedi punto 3).
- Tabella 3.4 "Entropy DQN"

  - "entropy_coeff": peso che viene applicato all'entropia quando viene calcolata, prima di utilizzarla nelle operazioni successive (vedi entropy_coeff).

Per quanto riguarda le simulazioni eseguite con iperparametri con valori differenti da quelli standard appena descritti si è provato:

- "exploration_fraction" maggiore dello standard 0.1, per giochi più complessi può rivelarsi utile aumentare il periodo di esplorazione. È da notare che "exploration_fraction" indica per quale percentuale del numero totale di mosse esplorare, l'esplorazione non dipende solo da questa variabile ma anche appunto dal numero totale di mosse. Prendiamo ad esempio due simulazioni lanciate con lo stesso "exploration_fraction" pari a 0.1, cioè entrambe useranno il primo 10% delle mosse totali per esplorare. Le mosse totali sono un altro dei parametri modificabili, se la prima simulazione in totale eseguirà $1K$ mosse, significa che esplorerà per le prime 100 (il 10% di $1K$), se la seconda invece eseguirà $1M$ di mosse, le mosse che spenderà ad esplorare saranno $100K$ (il 10% di $1M$).

- "exploration_final_eps", che controlla invece il valore di ε al termine della fase esplorativa è stato testato con valori diversi solo per Stochastic DQN e Entropy DQN.

- Sono state fatte molte prove cambiando il numero totale di mosse ("total_timesteps"), che come spiegato sopra influisce indirettamente anche su quanto l'algoritmo esplora.

- "dueling" e "param_noise" attivano due modifiche a DQN compatibili con lower bound DQN, e sono quindi state provate in congiunzione con la modifica introdotta da me.

- "prioritized_replay" non è compatibile con lower bound DQN, e quindi l'unica prova eseguita in questo caso è stata un confronto fra lower bound DQN e DQN originale con "prioritized_replay" attivo.

- Sono state eseguite anche simulazioni con più iperparametri modificati contemporaneamente, ma saranno descritte più in dettaglio nel capitolo relativo ai test.

3.7 Tecnologie utilizzate

Questo progetto è stato sviluppato con l'ausilio di numerose tecnologie che hanno consentito e semplificato lo svolgimento delle diverse fasi di realizzazione e prova delle modifiche apportate a DQN.

In un primo momento, durante la pianificazione, si è optato per addestrare e testare la rete neurale contro i giochi Atari. La scelta è dunque ricaduta sul principale e più noto strumento che consente di svolgere tale operazione, fornendo implementazioni di diverse versioni dei giochi interfacciabili con le NN, ovvero OpenAI Gym [6]. Più in generale, tale framework offre una vasta gamma di ambienti (sfide e problemi), tra i quali si annoverano numerosi test ormai classici di intelligenza artificiale, contro i quali mettere alla prova i propri agenti e valutare le eventuali modifiche apportate (trattato nella sezione 4.1 in quanto toolkit per Atari 2600).

Per quanto concerne, poi, il reperimento di Deep Q-Network, si è deciso di utilizzare la piattaforma OpenAI Baselines [8], che garantisce l'accesso al codice di svariati algoritmi mirati alla risoluzione di problemi di Reinforcement Learning (tra essi figura l'implementazione di DQN da cui il presente lavoro prende le mosse). Ad orientare la decisione è stata una considerazione di natura pratica: la medesima organizzazione, OpenAI (fondata nel 2015 per condurre ricerche nell'ambito dell'intelligenza artificiale) mette a disposizione entrambi gli strumenti menzionati sopra. Ciò significa che gli algoritmi di OpenAI Baselines contengono già il codice necessario per interfacciarsi in modo diretto con i giochi di OpenAI Gym, senza che sia l'utente a doverlo scrivere.

Occorre precisare che la rete neurale della versione di DQN contenuta in OpenAI Baselines sfrutta TensorFlow [1], la libreria open source di riferimen-

to per la creazione e lo sviluppo di modelli di Machine Learning. I creatori di TF hanno altresì sviluppato uno strumento addizionale denominato TensorBoard [17], adoperato per visualizzare graficamente informazioni relative alla rete neurale ed all'evoluzione della computazione, come si illustrerà in seguito.

Infine, in considerazione della mole di test e simulazioni previsti, si è optato per utilizzare Colaboratory [26], un notebook Jupyter disponibile gratuitamente che permette l'esecuzione di codice in cloud. Oltre a disporre della potenza di calcolo necessaria, Colaboratory (colab) offre la possibilità di avvalersi di GPU per velocizzare le operazioni.

3.7.1 TensorBoard

TensorBoard è uno strumento web messo a disposizione dai creatori di TensorFlow come elemento di supporto alla creazione di reti neurali: consente infatti di visualizzare graficamente alcuni dei calcoli della NN in tempo reale, di realizzare grafici dell'evoluzione di variabili scelte dall'utente e di poter osservare ed esaminare su più livelli il grafo della computazione. Tutto ciò facilita il debugging e l'ottimizzazione della rete.

Occorre introdurre una precisazione: a differenza delle modifiche di DQN illustrate in questo capitolo, TensorBoard non implica alcuna variazione nel funzionamento dell'algoritmo originale, dal momento che si configura semplicemente come un meccanismo volto all'ottenimento di informazioni perlopiù grafiche.

In circostanze normali, ovvero quando i calcoli sono eseguiti localmente, permettere ad una NN di interfacciarsi con TensorBoard comporta una modifica relativamente semplice. Occorre infatti creare un tf.summary.FileWriter() e dargli in input il percorso della cartella di log e le informazioni relative a ciò che si vuole visualizzare con TensorBoard (ai fini della presente tesi si è scelto di visionare il grafo della computazione). Tale operazione presenta tuttavia una serie di complicazioni quando l'esecuzione dell'elaborazione avviene in remoto, come nel caso di questo progetto. Dal momento che si è

deciso di importare il programma su un foglio Google Colaboratory, i dati che il *FileWriter* mette nella cartella di log e la cartella stessa si trovano sul server remoto di colab. TensorBoard di conseguenza non può accedere ad essi in modo diretto: per ovviare a questo problema si è utilizzato *ngrok* [12], un programma in grado di stabilire un collegamento tra TensorBoard e la cartella di log.

Capitolo 4

Atari 2600

Un metodo fondamentale per confrontare in modo significativo algoritmi differenti ed estremamente complessi consiste nel testarli in ambienti comuni e standardizzati. Proprio per questo motivo addestramenti e prove sono stati eseguiti sui giochi Atari.

Atari 2600 è una console di gioco creata nel 1977, per la quale nel corso degli anni sono stati realizzati numerosi giochi, anche da altre società o da singole persone. Questi giochi sono interessanti dal punto di vista del reinforcement learning per la quantità degli stessi, l'ampiezza e copertura del range di difficoltà, e la diversità di generi (action, strategy, racing, sport, adventure ecc...).

Per questi motivi in passato sono stati ampiamente utilizzati come ambiente di prova per algoritmi di RL e ormai possono essere considerati uno standard in questo campo di ricerca.

Per la presente tesi è stato usato un sottoinsieme dei giochi Atari 2600. Per motivi di tempo e potenza di calcolo, non è stato possibile testare le versioni di DQN realizzate contro tutti i giochi, per questo il livello di difficoltà ha rappresentato il principale criterio di selezione (per difficoltà in questo caso si intende principalmente la sparsità delle ricompense, perché meno informazioni ha l'algoritmo, più difficile risulta apprendere strategie efficaci). Dato che il focus di questo progetto è proprio quello di migliorare i risultati

in caso di problemi con ricompense sparse sono stati scelti i giochi in cui, nei paper esaminati, si ottenevano i punteggi più bassi:

- Alien
- Amidar
- Asteroids
- Berzerk
- Breakout
- Centipede
- Chopper Command
- Frostbite
- Gravitar
- H.E.R.O.
- Montezuma's Revenge
- Ms. Pac-Man
- Pitfall!
- Private Eye
- Seaquest
- Solaris
- Venture
- Yars' Revenge

4.1 OpenAI Gym

OpenAI Gym [6] è un toolkit per lo sviluppo e il confronto di algoritmi di reinforcement learning.

In una descrizione molto astratta il problema tipico di RL si compone di un ambiente e di un agente. L'agente può osservare lo stato dell'ambiente e compiere azioni, alle azioni l'ambiente risponde cambiando il proprio stato e fornendo una eventuale reward all'agente. OpenAI Gym fornisce l'ambiente, a scelta fra una grande varietà di problemi diversi, in cui addestrare i propri agenti (ovvero gli algoritmi di RL).

Le categorie di problemi attualmente disponibili su OpenAI Gym sono:

- Toy text e Classic control: ambienti testuali o classici dalla letteratura del reinforcement learning, tipicamente semplici, usati come introduzione all'argomento.

- Algorithmic: tipici task per calcolatrici e computer; addizioni, moltiplicazioni, operazioni su sequenze di numeri. In questo tipo di ambienti gli agenti devono imparare dagli esempi quali calcoli eseguire.

- Atari: i giochi Atari 2600 già discussi precedentemente, anche tramite l'integrazione di Arcade Learning Environment.

- 2D and 3D robots: task di controllo motorio di robot. Si appoggiano al motore fisico MuJoCo, realizzato proprio per questo tipo di simulazioni.

Per questo progetto ho usato OpenAi Gym per il fatto che implementa i giochi Atari, è direttamente compatibile con OpenAI Baselines (punto di partenza per lo sviluppo dell'algoritmo) ed è uno dei toolkit più utilizzati per questo scopo (quindi i risultati ottenuti sono direttamente confrontabili con molte altre ricerche).

Per assicurarsi che un algoritmo impari effettivamente a risolvere il problema su cui si addestra, invece che memorizzare una sequenza di mosse efficaci, sono state usate varie strategie che cercano di rendere l'ambiente ogni volta diverso o non totalmente prevedibile.

- "No-op starts" consiste nel far passare alcuni frame (numero casuale calcolato ogni volta) all'inizio di ogni episodio prima che l'agente possa osservare e agire. In questo modo se l'ambiente cambia con il tempo, ogni volta sarà leggermente diverso (ad esempio posizione dei nemici o tempismo delle trappole).

- "Human starts" fa cominciare ogni episodio da uno stato scelto casualmente da un insieme di stati di gioco ottenuti da giocatori umani. In questo caso le condizioni iniziali sono ancora più differenziate che nel caso precedente, ma richiede di avere una quantità sufficiente di stati memorizzati.

- "Sticky actions" rende incerta l'azione scelta dall'agente: ogni volta che l'agente decide un'azione c'è il 25% di probabilità che al suo posto venga

eseguita l'ultima azione scelta precedentemente. Anche in questo modo si scoraggia l'algoritmo ad imparare una sequenza di azioni, perché ogni volta il risultato di quello stesso gruppo di mosse potrebbe essere differente.

Ognuno dei giochi Atari è disponibile in più versioni differenti: "v0", "v4", "Deterministic", "NoFrameskip" e "ram".

"v0" indica che sono utilizzate le sticky actions contrapposto a "v4" che indica azioni normali. "Deterministic" significa che si applica un frameskip fisso di 4, con "NoFrameskip" non viene applicato nessun frameskip, mentre se non è usato nessuno di questi due argomenti, si ha un frameskip casuale di 2, 3 o 4. Come discusso in precedenza normalmente agli algoritmi viene fornito in input uno o più frame del gioco per decidere l'azione successiva. Utilizzando "ram", invece, l'input fornito all'algoritmo è il contenuto delle celle di ram che riguardano lo stato del gioco.

Capitolo 5

Test e Risultati

In questo capitolo esaminiamo in dettaglio le prove effettuate ed i relativi risultati ottenuti. Dato che per questa tesi sono state implementate diverse modifiche di DQN, vedremo i test di ognuna per poi confrontare le performance fra loro e con algoritmi di riferimento.

I fogli colab utilizzati per eseguire le simulazioni hanno una certa potenza di calcolo e un limite massimo di tempo sulla lunghezza delle simulazioni (circa 12 ore). Inizialmente per ogni test eseguito su una delle mie versioni modificate, effettuavo contemporaneamente su un foglio colab separato un test con gli stessi iperparametri, ma utilizzando come algoritmo DQN originale, in modo da avere un paragone diretto per i risultati. Considerando le restrizioni di colab, il limite apparentemente sembrava di $2M$ di step per entrambi i test. Successivamente ho notato che la potenza di calcolo fornita non era associata ad ogni foglio colab, ma all'utente, quindi eseguendo due simulazioni contemporaneamente entrambe risultavano molto rallentate. Da quel momento le simulazioni sono state eseguite solo in sequenza e questo ha permesso di raggiungere, nello stesso tempo limite di 12 ore, un numero di step doppio ($4M$ di step per simulazione). Per questo motivo alcuni dei test iniziali terminano a $2M$ di passi.

Tipicamente i risultati ottenuti vengono normalizzati e trasformati in percentuali, in cui il 100% indica il punteggio di un esperto umano, secondo

la formula riportata di seguito:

$$100 \times \frac{\text{DQN score} - \text{random play score}}{\text{human score} - \text{random play score}} \tag{5.1}$$

Nei test per questa tesi, tuttavia, si dispone di un confronto diretto dato che la comparazione avviene per simulazioni equivalenti per ambienti e iperparametri e differenti solo per l'algoritmo. I punteggi mostrati per le simulazioni sono quindi quelli originali ("DQN score" nella formula).

5.1 Lower Bound DQN

Lower Bound DQN è la modifica testata in modo più estensivo dato che inizialmente, a parte alcuni miglioramenti successivi, sembrava essere il focus della tesi. Invece sono state introdotte le versioni Stochastic DQN e Entropy DQN, ma a causa della durata delle prove non è stato possibile eseguire una quantità di test paragonabile a quelli eseguiti su LB DQN.

Per prima cosa vediamo i risultati ottenuti con gli iperparametri di default:

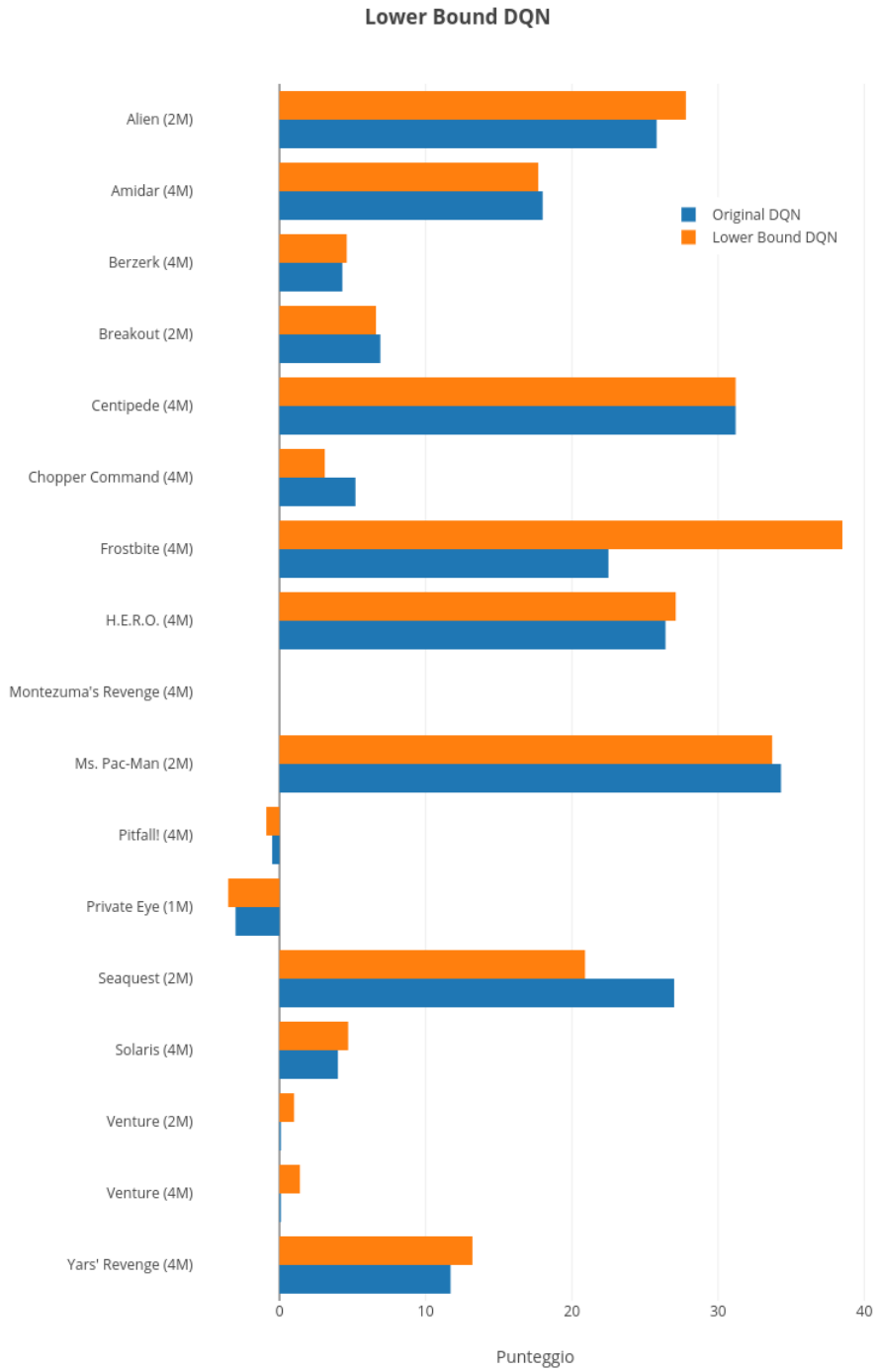

Figura 5.1: Risultati ottenuti da LB DQN e DQN originale; il numero tra parentesi indica quanti milioni di step sono stati usati per ogni addestramento.

Per le motivazioni spiegate in precedenza alcune simulazioni sono state eseguite con un numero di step minore rispetto al massimo raggiungibile in queste condizioni ($4M$ di passi). Di seguito il grafico complessivo di confronto.

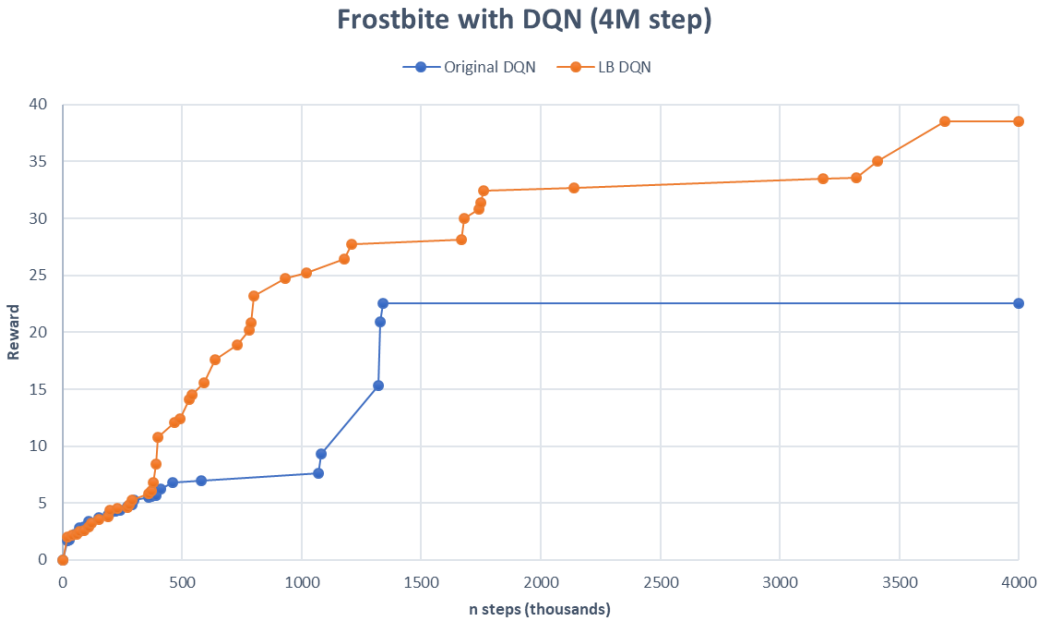

Figura 5.2: LB DQN (arancione) e DQN originale (blu) a confronto in Frostbite (Atari 2600)

Sono particolarmente notevoli i risultati realizzati in Frostbite e Venture dei quali allego il grafico di dettaglio di confronto (LB DQN contro DQN originale). Nei risultati generali Venture è piuttosto basso come punteggio, ma paragonando i due algoritmi solo su questo gioco, LB DQN mostra un miglioramento molto marcato. La modifica in questo caso particolare sembra addirittura fare la differenza fra migliorare col tempo e non apprendere niente.

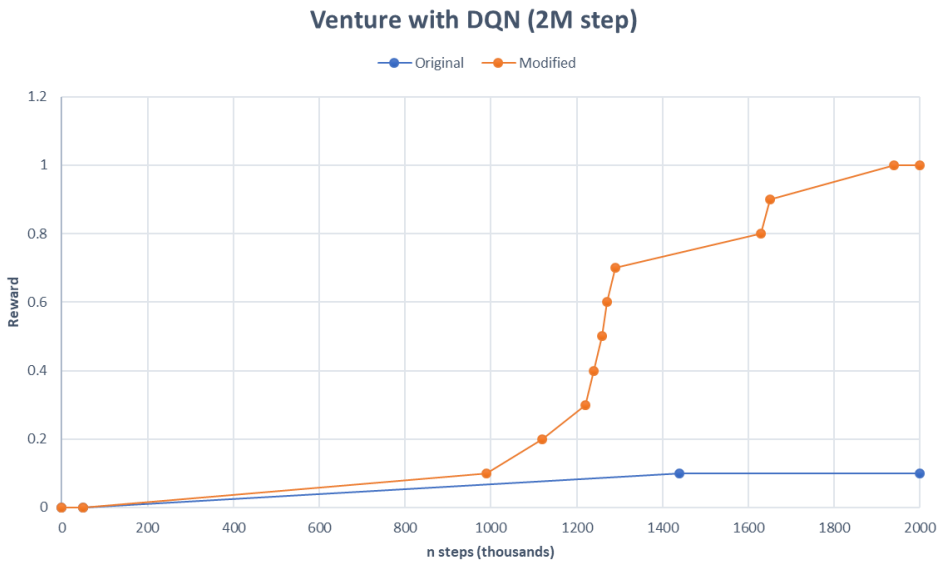

Figura 5.3: LB DQN (arancione) e DQN originale (blu) a confronto in Venture (Atari 2600). Simulazione da 2M di passi.

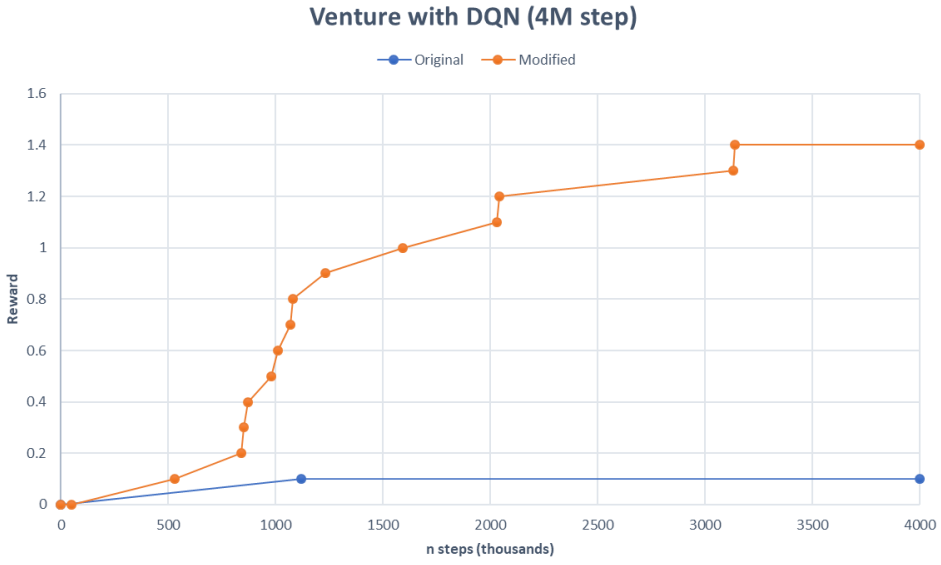

Figura 5.4: LB DQN (arancione) e DQN originale (blu) a confronto in Venture (Atari 2600). Simulazione da 4M di passi.

Gli unici altri due giochi in cui il Lower Bound Buffer sembra dare miglioramenti, anche se più modesti dei casi discussi sopra, sono Solaris e Yars'

Revenge.

Vediamo ora test eseguiti con Dueling DQN (iperparametro "dueling" settato a True), modifica attivata sia per l'originale che per la versione con Lower Bound:

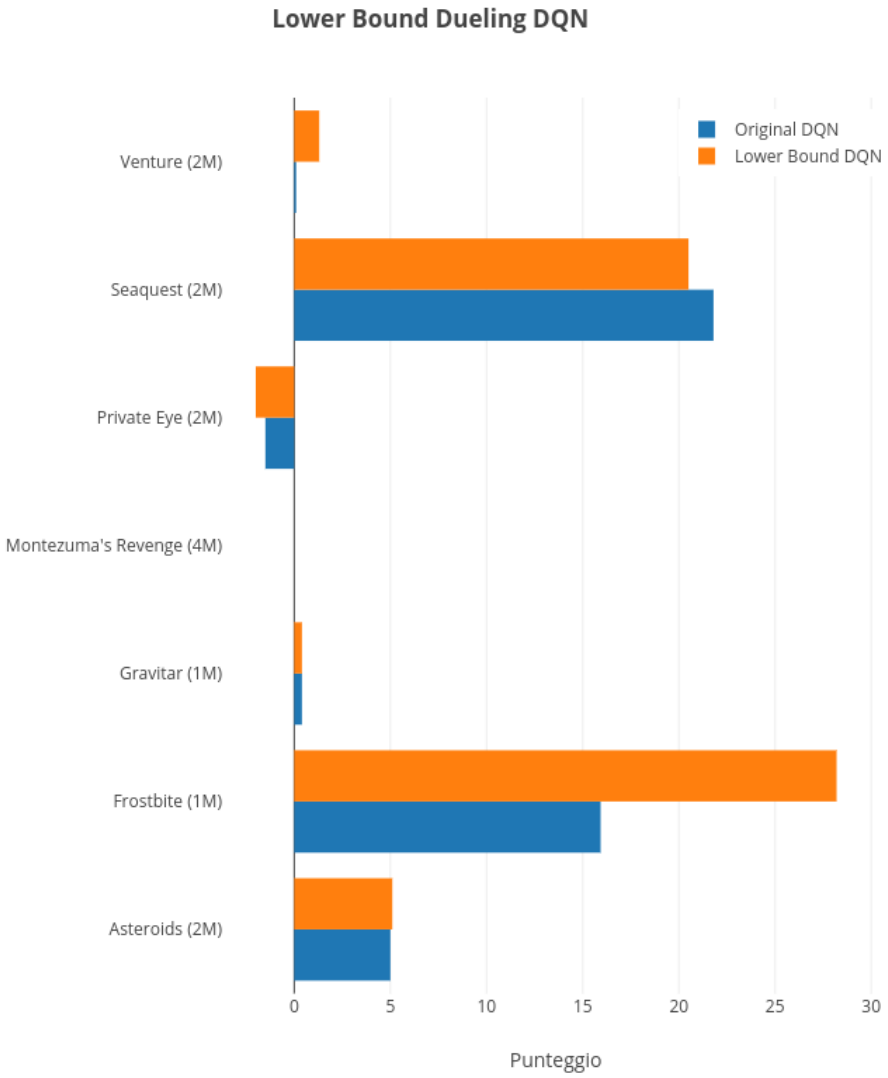

Figura 5.5: Risultati ottenuti da LB Dueling DQN e Dueling DQN; il numero tra parentesi indica quanti milioni di step sono stati usati per ogni addestramento.

Dai test eseguiti sembra emergere che questa modifica in generale migliora

i risultati in entrambi i casi, mantenendo le differenze relative inalterate. Ad esempio in Frostbite LB Dueling DQN ottiene un punteggio circa 50% maggiore di Dueling DQN, proprio come nel caso senza Dueling attivo: i punteggi erano diversi, ma duello di LB DQN era più o meno superiore del 50% al punteggio di DQN originale.

In Frostbite sono state eseguite simulazioni con Parameter Space Noise e anche con Parameter Space Noise e Dueling contemporaneamente. Come si può osservare dai grafici tuttavia, con la configurazione utilizzata, questa modifica ha ottenuto effetti negativi invece di migliorare i punteggi, sia per DQN originale, sia per LB DQN.

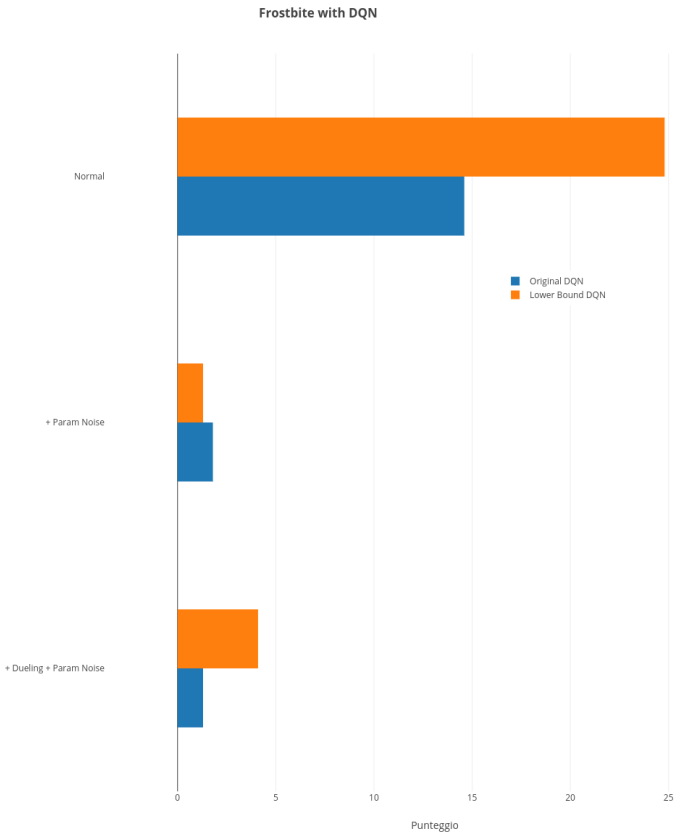

Figura 5.6: Risultati ottenuti da LB DQN e DQN originale con combinazioni di Parameter Space Noise e Dueling attive. Tutte le simulazioni usate sono ottenute con $1M$ di step di addestramento.

Prioritized Replay, una delle varianti di DQN preesistente a questa tesi, non è compatibile con LB DQN per cui, a differenza di Dueling e Parameter Space Noise, è stata utilizzata solo come "avversario" per confrontare i risultati dei test. Stranamente Prioritized Replay ha ottenuto un punteggio addirittura inferiore a DQN originale: questo era inaspettato perché dal paper "Prioritized experience replay" [28] sembrava apportare un miglioramento notevole. Questo risultato negativo potrebbe tuttavia essere dovuto al particolare ambiente (Frostbite) o agli iperparametri utilizzati non adeguati a questa variante.

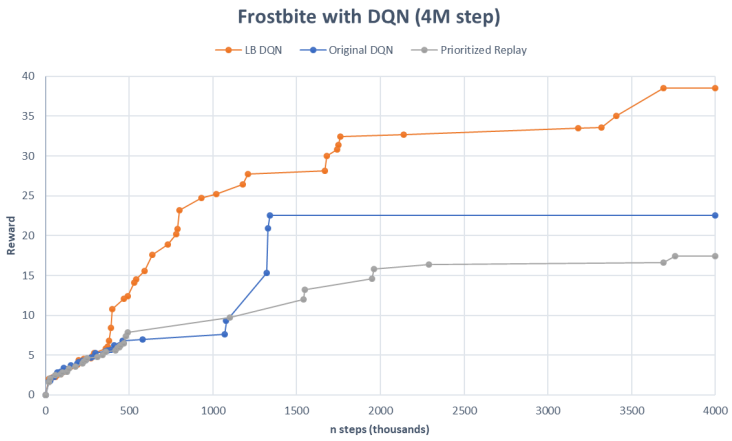

Figura 5.7: LB DQN e DQN originale e Prioritized Replay a confronto. Le simulazioni usate sono ottenute con $4M$ di step di addestramento su Frostbite.

Dopo l'implementazione delle funzioni di salvataggio e caricamento descritte nella sezione 3.3, è stata effettuata una simulazione da $20M$ di step secondo questo schema:

1. si eseguono $4M$ di passi
2. si salva lo stato (pesi della rete, iperparametri, contenuto dei buffer, altre variabili) in un unico file
3. il file viene copiato su google drive

4. dopo la chiusura della prima macchina virtuale, ne viene aperta una seconda

5. si ricarica la simulazione interrotta sulla seconda VM usando le informazioni contenute nel file

6. si riprende l'addestramento (si riparte dal punto 1.)

Il risultato mostra alcune criticità, rappresentate graficamente da picchi negativi sull'asse delle reward, che probabilmente indicano la necessità di dover considerare anche altri elementi aggiuntivi al momento del salvataggio delle informazioni: in corrispondenza di ogni interruzione e ripresa dell'addestramento (ogni $4M$ di step) si rileva un temporaneo ma significativo calo di prestazioni dell'algoritmo.

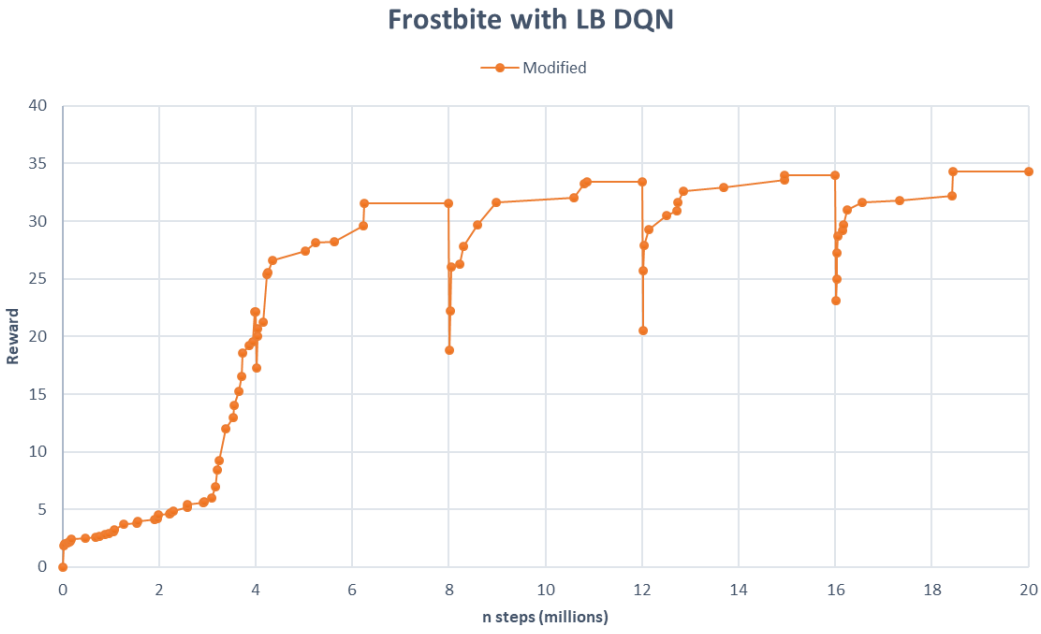

Figura 5.8: Risultati ottenuti da LB DQN in Frostbite con simulazione da $20M$ di step.

5.2 Stochastic DQN

La realizzazione di questa modifica ha richiesto molte decisioni e prove perché, a parte le difficoltà tecniche, come emerge dalla spiegazione nella relativa sezione (3.4), non esiste un solo modo "corretto" di implementazione. I primi risultati non sono stati molto incoraggianti, e il risultato finale non raggiunge LB DQN, ma è comunque significativamente migliore di DQN originale, e almeno teoricamente è più robusto di LB DQN. Occorrerebbe ampliare il numero di ambienti e la quantità delle prove e la loro specificità per poter trarre conclusioni più accurate.

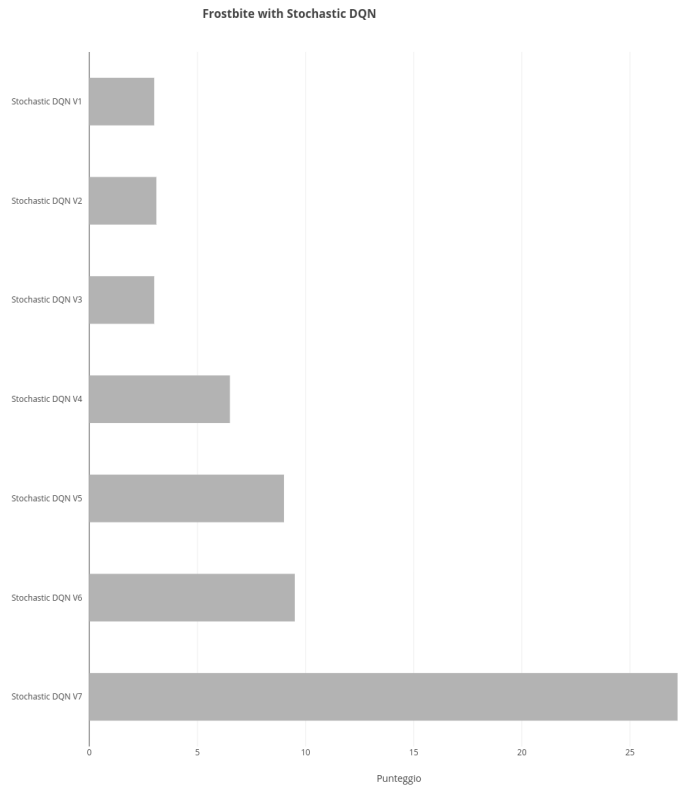

Figura 5.9: Risultati ottenuti da Stochastic DQN in Frostbite con simulazioni da $4M$ di step.

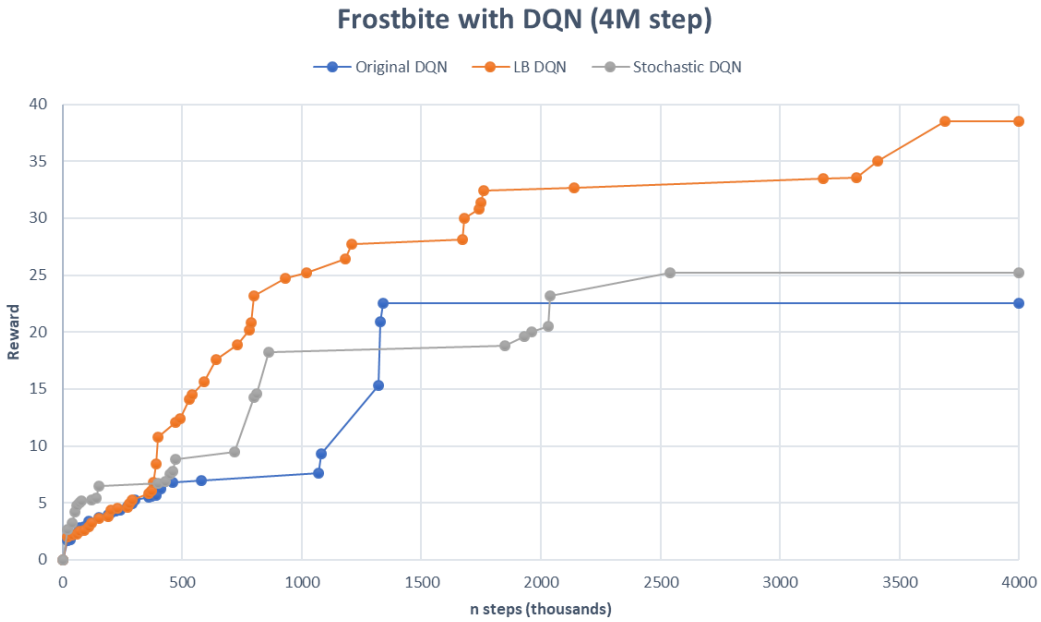

Figura 5.10: Comparazione fra Stochastic DQN, LB DQN e Original DQN in Frostbite con simulazioni da $4M$ di step.

5.3 Entropy DQN

Dato che Entropy DQN si basa su Stochastic DQN (ha bisogno di probabilità associate ad ogni possibile azione per calcolare l'entropia), per ogni variante di quest'ultimo potremmo avere una corrispondente variante di Entropy DQN. Si è scelto di utilizzare direttamente la versione finale di Stochastic DQN, cioè quella che ha mostrato i risultati migliori.

Inizialmente vengono paragonate fra loro le due implementazioni di Entropy DQN realizzate, quella in cui le azioni vengono scelte come in DQN originale e quella che usa le probabilità come Stochastic DQN.

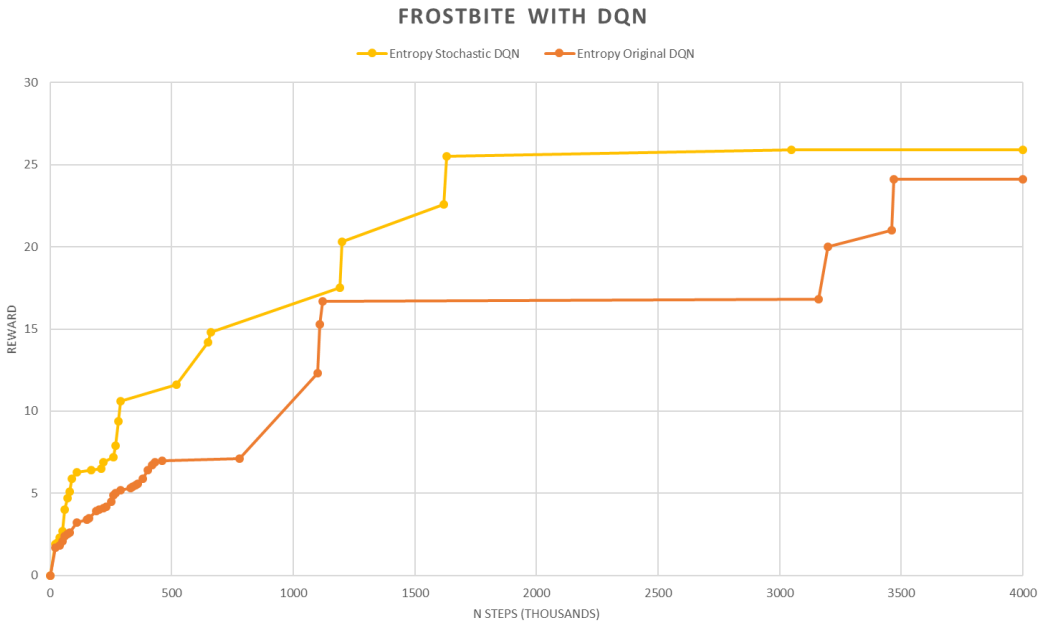

Figura 5.11: Confronto fra le due varianti di Entropy DQN. Simulazioni in Frostbite con $4M$ di step.

Infine, dopo aver esaminato separatamente ogni modifica realizzata, vengono confrontati in un grafico complessivo i risultati ottenuti da ogni variante; inoltre nel grafico è rappresentato anche DQN originale come caso base.

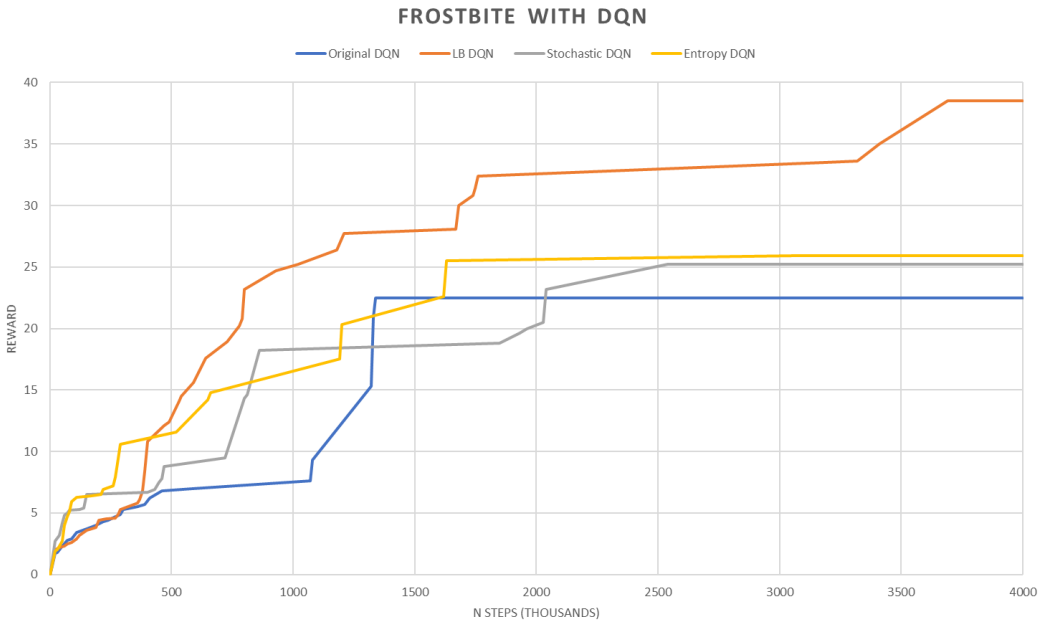

Figura 5.12: Confronto diretto fra tutte le modifiche implementate e DQN originale. Simulazioni in Frostbite con $4M$ di step.

La modifica complessivamente migliore (ricordiamo che il grafico mostra i risultati su Frostbite) si è rivelata essere LB DQN, ma anche Stochastic DQN e Entropy DQN esibiscono punteggi superiori a DQN originale, in modo stabile.

Capitolo 6

Difficoltà incontrate

Tra le difficoltà riscontrate durante il corso della tesi figurano i limiti imposti da Colaboratory: una computazione relativamente lenta e la chiusura della macchina virtuale ogni 12 ore, indipendentemente dalla presenza di operazioni in corso. Questo ha limitato la dimensione delle simulazioni a circa $4M$ di passi; per eseguire test più lunghi è stato necessario modificare il codice, così da poter salvare lo stato di una simulazione e caricarlo successivamente per continuare l'addestramento. Oltre a salvare localmente lo stato della simulazione, è anche necessario scaricarlo, perché quando la macchina virtuale chiude, tutti i file locali vengono eliminati.

Con la suddivisione di uno stesso addestramento in più sezioni, si è evidenziato come conseguenza delle interruzioni un problema di temporanea diminuzione della capacità di gioco della rete all'inizio di ogni fase. La capacità di gioco della rete torna comunque rapidamente al valore ottenuto alla fine della fase precedente, e successivamente la rete riprende ad imparare normalmente.

Un altro problema si è manifestato durante i test delle versioni iniziali, ottenute modificando DNQ, con l'aumento dei tempi di computazione: ogni step richiedeva sempre più tempo del precedente, tanto che nelle 12 ore limite non venivano completati nemmeno $1M$ di passi. Ho poi riscontroato che questo problema era legato al modo di eseguire il test sulle esperienze del

lower bound buffer, ed è rimasto presente fino alla sostituzione di quella policy con quella utilizzata attualmente.

Un aspetto negativo dell'ultima e migliore versione di Lower Bound DQN è rappresentato dal fatto che il Lower Bound buffer, nel corso di ogni addestramento, viene utilizzato sempre meno (cioè sempre più tuple non passano il test), fino ad essere quasi totalmente ignorato. Tuttavia questo non è un comportamento inatteso: man mano che l'algoritmo impara in modo sempre più accurato le mosse e strategie migliori, meno rilevanza avranno le informazioni sui valori minimi delle mosse, perché appunto LB DQN calcolerà una stima ancora più accurata.

Una difficoltà più generale si è rivelata essere la lunghezza delle computazioni. Questa rende lento testare modifiche al codice: se per ottenere i risultati serve un'intera simulazione (più di 8 ore), per ogni modifica è necessario quindi attendere molto tempo per verificarne la correttezza e l'efficacia.

Infine, un altro problema significativo è stato dato dal fatto che OpenAI Baselines (codice fornito da OpenAI che implementa i maggiori algoritmi di RL), per quanto comodo, presenta problemi di bug, formattazione e documentazione. Questi problemi sono così significativi che è stato addirittura creato "Stable Baselines" [11], un progetto che partendo da OpenAI Baselines lascia quasi inalterato l'aspetto funzionale, ma ristruttura molto del codice presente. Questo ha comportato alcune difficoltà aggiuntive, in particolare per le modifiche su Entropy DQN, dato che in questa versione occorreva eseguire piccole modifiche specifiche in parti di codice molto complesse, intricate e scarsamente documentate.

Conclusioni

L'utilizzo di reti neurali per la risoluzione di problemi di intelligenza artificiale, in particolare di reinforcement learning, si è rivelato molto utile ed interessante. Molte delle problematiche insorte nel corso di questo progetto risultano piuttosto specifiche ed emergono soltanto quando si lavora in maniera più approfondita nel campo delle reti neurali. Soprattutto per questo l'esperienza si è rivelata proficua: nel contesto accademico mi ha infatti permesso di apportare il mio contributo ad un ramo di studi in continua evoluzione, offrendomi al contempo l'opportunità di cimentarmi nella risoluzione di problemi che non sempre vengono affrontati durante i corsi universitari.

Questa tesi, che è stata svolta in continuità con la precedente attività di tirocinio, ha rappresentato per me un'occasione per approfondire un argomento che reputo di grande interesse ed attualità, anche in vista di eventuali opportunità lavorative future in questo stesso ambito.

6.1 Sviluppi futuri

- Come già illustrato, tenuto conto dei risultati positivi delle varianti Stochastic DQN e Entropy DQN, riterrei opportuno accumulare un numero di test funzionali (su più giochi Atari e con iperparametri differenti) che permettano un'analisi più approfondita ed accurata.

- Preso atto dei miglioramenti introdotti da LB DQN e Entropy DQN, in modo ortogonale (cioè indipendentemente l'uno dall'altro), sembra

ragionevole proporre la realizzazione di una nuova versione che unisca le due modifiche precedenti.

- Riscontrato sperimentalmente il problema legato alla riduzione nell'utilizzo di LB buffer all'avanzare dell'allenamento, occorrerebbe prospettare un'adeguata soluzione. Per questo ipotizzo due possibilità alternative: progettare e realizzare un metodo per continuare ad inserire esperienze significative nel LB buffer o, per evitare un inutile overhead, terminare l'utilizzo del LB buffer quando diventa inefficiente (ad esempio al raggiungimento di una soglia massima di esperienze scartate).

- Resta da esaminare più approfonditamente un caso particolare, soprattutto in considerazione dell'elevata difficoltà legata all'ambiente specifico: si tratta di Montezuma's Revenge. Con un numero sufficientemente elevato di passi casuali DQN raggiunge la prima ricompensa, senza però riuscire a ripetersi in modo significativo. Con LB DQN, sfruttando il Lower Bound buffer occorre verificare se, come si ipotizza, l'algoritmo sia in grado di raggiungere la prima ricompensa in modo consistente.

- Modifica a DQN originale:

  - con l'obbiettivo di massimizzare "exploration" si può pensare di introdurre un buffer addizionale in cui accumulare esperienze generate da una seconda rete neurale, ricordando che DQN può apprendere anche da esperienze ottenute con policy diverse dalla propria.

  - In questo caso la seconda rete neurale che alimenta il buffer addizionale ha l'unico scopo di esplorare, possibilmente in modo più rispetto alla classica scelta casuale. Per raggiungere tale obbiettivo si potrebbe testare un sistema di ricompense indipendente dalle reward di gioco, ad esempio premiante in funzione della scoperta di nuovi stati e penalizzante in caso di "morte".

– Effettuate tali modifiche, DQN dovrebbe addestrarsi estraendo esperienze da entrambi i buffer. Sarà necessario considerare anche un nuovo ε che renda possibile controllare il grado di esplorazione stabilendo il rapporto fra le esperienze estratte dai due buffer.

Appendice A

Definizioni e altri termini di frequente utilizzo

Nel corso di questo trattato sono stati utilizzati alcuni termini specifici per dei quali di seguito definiamo i principali e più frequentemente utilizzati, che non siano già stati definiti nel corpo della tesi.

- Episodio/partita: si intende un intero tentativo di risolvere la sfida presentata, generalmente composto di molte azioni. "Partita" viene usato come sinonimo perché per questa tesi l'ambiente è sempre uno dei giochi Atari, e quindi episodio e partita indicano la stessa entità.
- On-policy/off-policy: i primi metodi hanno una sola policy usata sia per scegliere azioni che per la valutazione e miglioramento tramite esperienze. I secondi hanno due policy distinte, una per compiere le decisioni, l'altra che apprende dai risultati.
- model-dependent: proprietà di algoritmi che al loro interno possiedono un modello di riferimento dell'ambiente che permette la previsione dei risultati delle azioni da compiere.
- model-free: opposta a model dependent, indica algoritmi che cercano soluzioni senza conoscenze preimpostate dell'ambiente, solo tramite l'accumulo di esperienze.

Bibliografia

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: `1603.04467 [cs.DC]`.

[2] Zafarali Ahmed et al. "Understanding the impact of entropy in policy learning". In: *arXiv preprint arXiv:1811.11214* (2018).

[3] M. G. Bellemare et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents". In: *Journal of Artificial Intelligence Research* 47 (giu. 2013), pp. 253–279.

[4] Marc G Bellemare, Will Dabney e Rémi Munos. "A distributional perspective on reinforcement learning". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 449–458.

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

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

[7] *Cloudpickle*. 2019. URL: `https://github.com/cloudpipe/cloudpickle`.

[8] Prafulla Dhariwal et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[9] Scott Fujimoto, Herke van Hoof e David Meger. "Addressing function approximation error in actor-critic methods". In: *arXiv preprint arXiv:1802.09477* (2018).

[10] Tuomas Haarnoja et al. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *arXiv preprint arXiv:1801.01290* (2018).

[11] Ashley Hill et al. *Stable Baselines*. `https://github.com/hill-a/stable-baselines`. 2018.

[12] @inconshreveable. *Ngrok*. 2019. URL: `https://ngrok.com/`.

[13] 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`. URL: `http://dx.doi.org/10.1126/science.aau6249`.

[14] 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`.

[15] Andreas Kaplan e Michael Haenlein. "Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence". In: *Business Horizons* 62.1 (2019), pp. 15–25. ISSN: 0007-6813. DOI: `https://doi.org/10.1016/j.bushor.2018.08.004`. URL: `http://www.sciencedirect.com/science/article/pii/S0007681318301393`.

[16] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[17] D Mané et al. *TensorBoard: TensorFlow's visualization toolkit, 2015*.

[18] Tom M Mitchell. *Machine learning*. 1997.

[19] Volodymyr Mnih et al. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. 2016, pp. 1928–1937.

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

[21] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[22] Ofir Nachum et al. “Data-efficient hierarchical reinforcement learning”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 3303–3313.

[23] Ofir Nachum et al. “Near-Optimal Representation Learning for Hierarchical Reinforcement Learning”. In: *arXiv preprint arXiv:1810.01257* (2018).

[24] Ashvin Nair et al. “Overcoming exploration in reinforcement learning with demonstrations”. In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2018, pp. 6292–6299.

[25] Matthias Plappert et al. “Parameter space noise for exploration”. In: *arXiv preprint arXiv:1706.01905* (2017).

[26] research.google.com. *Welcome to Colaboratory*. Mag. 2018. URL: `https://colab.research.google.com/notebooks/welcome.ipynb`.

[27] 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.

[28] Tom Schaul et al. “Prioritized experience replay”. In: *arXiv preprint arXiv:1511.05952* (2015).

[29] Richard S Sutton e Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[30] Hado Van Hasselt, Arthur Guez e David Silver. “Deep reinforcement learning with double q-learning”. In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[31] Weixun Wang et al. *Towards Cooperation in Sequential Prisoner’s Dilemmas: a Deep Multiagent Reinforcement Learning Approach*. 2018. arXiv: `1803.00162 [cs.AI]`.

[32] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *arXiv preprint arXiv:1511.06581* (2015).

[33] Ziyu Wang et al. “Sample efficient actor-critic with experience replay”. In: *arXiv preprint arXiv:1611.01224* (2016).

[34] Brian D Ziebart et al. “Maximum entropy inverse reinforcement learning”. In: (2008).

Ringraziamenti

Il presente lavoro è stato reso possibile grazie alla collaborazione del relatore, il professore Andrea Asperti, il cui apporto è stato prezioso e fondamentale.

Desidero ringraziare inoltre tutti coloro che hanno contribuito alla realizzazione di questa tesi: Daniele Cortesi, Davide Beretta e tutti gli altri compagni che hanno condiviso con me gli anni di università; la mia famiglia per l'aiuto nella stesura di questo lavoro fino alla fine e tutti gli amici che mi hanno sopportato quando parlavo di algoritmi e reti neurali.