Apprendimento per Rinforzo: Deep Q–Network e Discussione dei casi di Insuccesso

Maxim Gaina e Bartolomeo Lombardi

Università di Bologna*
{maxim.gaina, bartolomeo.lombardi}@studio.unibo.it

24 giugno 2017

Sommario

Dato l'algoritmo Deep Q–Network ideata e implementata da DeepMind, l'obiettivo di questo lavoro di progetto consiste nel comprendere per quale motivo tale agente, in breve DQN, ha ottenuto risultati sotto la soglia delle prestazioni umane su un sottoinsieme del parco giochi Atari 2600. Verrà quindi studiato l'Apprendimento per Rinforzo, l'algoritmo Q–Learning e la sua evoluzione DQN assieme all'architettura delle reti neurali sottostanti; in maniera aggiuntiva, pur non disponendo delle stesse risorse di calcolo, si cercheranno di fare delle prove pratiche in grado di mimare i risultati ottenuti precedentemente, analizzandoli. Infine verrà dato uno sguardo a quali sono state le innovazioni in questo ambito.

Indice

I Agente DQN **2**
- i Algoritmo Q–Learning 2
- ii Deep Q–Network 3
- iii Implementazione in LUA 5
  - iii.1 train_agent 5
  - iii.2 NeuralQLearner 5

II Reti Neurali Convoluzionali **7**
- i CNN nel contesto DQN 11
  - i.1 Preprocessing 11
  - i.2 Architettura del modello 11
  - i.3 Implementazione 11

III Game Over **11**
- i Alcune prove 13

IV Prestazioni scadenti: cause individuate **13**
- i Pianificazione a lungo termine . 14
- ii Scelte di carattere tattico 14
- iii Ricompense Ambigue ed elevata Complessità 15

V Sessioni di apprendimento **15**
- i *Breakout* 16
  - i.1 Rompere il muro 16
  - i.2 Situazione di stallo 16

VI Evoluzione dell'ambito RL **16**

VIIConclusioni **17**

*Progetto per il corso di Complementi di Linguaggi di Programmazione, A.A. 2016/2017, prof. Andrea Asperti.

Introduzione

Nell'ambito del *Machine Learning* si possono individuare diversi stili nella risoluzione dei problemi. L'apprendimento *supervisionato* prevede che a un modello vengano forniti esempi di input e relativi output, affinché esso individui delle regole in grado di mappare input futuri nella maniera più corretta possibile. Esistono diverse metriche in grado di individuare le prestazioni di un modello. L'apprendimento *non supervisionato*

invece, ammette che al modello vengano forniti dati non etichettati. Il suo compito sarà poi quello di individuare autonomamente dei pattern o delle features. Il tipo di apprendimento che viene preso in considerazione in questo progetto è l'apprendimento *per rinforzo* (*Reinforcement Learning, RL*). Si può dire che gli algoritmi RL interagiscono con ambienti dinamici in cui ci sono degli obiettivi da raggiungere, per poi ricevere un feedback sotto forma di *premio* o *punizione*. In altre parole, ci sono degli input e degli output etichettati come nell'apprendimento supervisionato, ma le etichette dei dati in output cambiano per adattarsi meglio all'ambiente.

Una tecnica per affrontare i problemi RL è l'algoritmo `Q-Learning` che, preso singolarmente, in alcuni casi presenta imperfezioni e persino comportamenti divergenti, sotto particolari condizioni (che verranno viste più tardi). È stato formalizzato e implementato dalla *start-up* DeepMind un agente in grado di imparare a giocare all'intero parco titoli di Atari 2600, tramite reti neurali deep che approssimano la funzione Q. L'algoritmo *Q–Learning* però, per risolvere i problema legati alla sua stabilità, è stato modificato inserendo il concetto di **Experience Replay** e altre modifiche che verranno citate più tardi. È necessario capire la motivazione fondamentale dietro a questi sforzi, cioè quella di creare un singolo algoritmo in grado di affrontare un'ampia varietà di problemi, che a sua volta è da sempre un obiettivo dell'Intelligenza Artificiale Generale. Un unico modello che affronta un intero parco giochi è un ottimo allenamento: per la maggior parte dei giochi Atari 2600 sono stati ottenuti risultati brillanti. Esiste tuttavia un insieme di giochi in cui l'agente *Deep Q–Network* (DQN) non supera le prestazioni umane o non migliora le metriche ottenute da metodi precedentemente usati. In questo progetto si prenderà uno dei giochi in cui l'agente DQN non si dimostra particolarmente bravo e si cercherà di capire perché.

La relazione è strutturata in questo modo:

- nella prima sezione verrà descritto l'algoritmo Q–Learning e le aggiunte portate dall'agente Deep Q–Network, dopo di che si passerà all'implementazione;
- la seconda sezione descriverà l'architettura delle reti neurali sottostante all'agente DQN e la relativa implementazione;
- verranno riportati gli esiti delle prove pratiche nella terza sezione;
- la quarta sezione tratterà le cause per cui un agente DQN possa ottenere scarsi risultati;
- infine si vedrà brevemente come si è mossa la ricerca che tratta questo tipo di apprendimento automatico.

I. Agente DQN

In questa sezione verrà descritto in breve l'algoritmo *Q–Learning* (QL), e in che modo l'agente DQN ne riprende il concetto e come quest'ultimo sia stato definito.

i. Algoritmo Q–Learning

L'algoritmo si basa su una funzione di questo tipo:

$$Q : S \times A \longrightarrow \mathbb{R},$$

dove S è l'insieme degli stati che può assumere l'ambiente dinamico e A sono le possibili azioni da eseguire. Dopo aver eseguito un'azione $a \in A$ l'agente si muove verso il prossimo stato ricevendo una ricompensa. Tuttavia, l'algoritmo deve imparare quale azione a sia *ottimale* per ogni stato, perché il suo obiettivo ultimo è quello di massimizzare la quantità totale di ricompense ricevute. Per azione ottimale si intende quella che a lungo termine ha la ricompensa più alta. Il QL è un algoritmo iterativo, sono quindi necessarie delle condizioni iniziali. L'apprendimento parte ritornando un valore predefinito, e ogni volta che agisce osserva un nuovo stato insieme alla ricompensa che possono dipendere dallo stato precedente e dall'azione selezionata, aggiornando il valore di Q. L'equazione 1 descrive l'algoritmo QL.

Learning rate α_t O anche *tasso di apprendimento*, viene usato per regolare quanto un nuovo

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha_t \cdot (r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{1}$$

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{2}$$

$$Q^*(s, a) = \max_\pi \mathbb{E}[R_t | s_t = s, a_t = a, \pi] \tag{3}$$

aggiornamento incide su quello che è già stato appreso, e si ha che $0 \leq \alpha_t \leq 1$. Infatti il valore 0 è un estremo che indica che l'algoritmo non memorizzerà nulla, mentre il valore 1 specifica che bisogna memorizzare solo le informazioni più recenti.

Discount factor γ Oppure *fattore di sconto*, che aiuta a determinare l'importanza delle ricompense ricevute. Analogamente a prima si ha che $0 \leq \gamma \leq 1$, dove lo zero imposta l'attenzione dell'algoritmo sulle ricompense immediate, mentre il valore 1 induce a scelte in funzione di ricompense migliori a lungo termine.

Ricompensa r_{t+1} Infine, r_{t+1} indica la ricompensa stessa dopo aver eseguito a dentro a s_t. La ricompensa è una sommatoria pesata dei valori attesi di ricompense ottenibili dagli **step** futuri a partire dallo stato corrente, e per i futuri Δt step i pesi sono calcolati da $\gamma^{\Delta t}$.

È detto **episodio** dell'algoritmo la sequenza di stati che termina con uno stato finale. In un videogioco l'episodio si può vedere come il tentativo di raggiungere un obbiettivo, o completare un livello, che può finire con fallimento o successo. Solitamente è necessario un ampio numero di episodi prima che l'agente sia in grado di fare qualcosa di accettabile. Le implementazioni dell'algoritmo QL sono varie, ma la forma più semplice si basa su una tabella in cui le righe possono essere gli stati e le colonne le azioni. Ogni cella $Q_{i,j}$ contiene il valore di quanto sia ottimale l'azione a_j nello stato s_i, e tale valore viene aggiornato opportunamente. Tuttavia, è un metodo non applicabile ad ambienti complessi in cui il numero degli stati è enorme. Un'altro modo è quello di approssimare la funzione tramite Reti Neurali Deep, ed è questa la strada che imbocca l'agente DQN che verrà visto fra poco.

Limiti di Q–Learning Anche se l'algoritmo QL appena accennato è stato prima introdotto e ne è stata poi dimostrata la convergenza, esso è conosciuto per il suo comportamento potenzialmente divergente se la funzione Q viene rappresentata da un approssimatore non lineare come le reti neurali. Come riportato in [1], uno dei problemi è che piccoli cambiamenti a Q possono cambiare significativamente la distribuzione dei dati, in più si aggiunge la correlazione fra sequenze di osservazione; insieme alla correlazione fra il vecchio valore $Q(s_t, a_t)$ e il nuovo valore appreso (nella letteratura spesso denominato come Q–target o **target value**) $r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$ facente parte dell'equazione 1. La correlazione tra valori consecutivi non porta a buoni risultati.

ii. Deep Q–Network

L'agente *Deep Q–Network* (DQN) ha come base l'algoritmo QL precedentemente descritto. QL infatti è fondato sulla versione iterativa dell'equazione di Bellman, che a sua volta si basa sulla seguente intuizione: se il valore ottimale di una sequenza di coppie *(azione, osservazione)* fosse conosciuta per ogni possibile azione prossima, allora la strategia ottimale è quella di scegliere l'azione che massimizza il valore atteso del valore appreso $r_{t+1} + \gamma \cdot \max_a Q(s_{t+1}, a)$. Le aggiunte fondamentali del DQN però sono le seguenti:

1. *Convolutional Neural Network* (CNN) per approssimare il massimo guadagno possibile seguendo una determinata politica π (distribuzione delle probabilità su azioni

da eseguire) 3, e dove viene fatto riferimento alla ricompensa in un determinato istante, R_t, definito dalla 2 dove T è il tempo di fine partita;

2. *Experience Replay*, un meccanismo che permette di allenare la CNN usando ricordi tratti da esperienze passate;

3. un nuovo metodo di aggiornamento dei valori di Q, usando un clone della rete neurale originaria.

Gli autori hanno anche dimostrato che togliendo anche uno solo di questi tre elementi i risultati peggiorano drasticamente. Seguiranno le spiegazioni più dettagliate dei cambiamenti apportati.

La funzione Q viene parametrizzata così diventando

$$Q(s, a; \Theta_i),$$

dove Θ_i sono i pesi della rete deep durante l'iterazione i-esima, e vengono aggiustati ad ogni iterazione per minimizzare l'errore quadratico nell'equazione di Bellman, dove le etichette vengono approssimate da

$$y = r + \gamma \max_{a'} Q(s', a'; \Theta_i^-),$$

con Θ_i^- che sono i parametri della rete deep allo step precedente (mantenuti appunti da un clone della rete). La rete deep in questione viene anche chiamata Q–Network. Come è già stato accennato, le etichette dipendono dai pesi della rete (al contrario di quanto si usa nell'apprendimento supervisionato), e la funzione costo (*loss function*) cambia ad ogni iterazione i. Senza riscrivere tutti i passaggi matematici della lettera originale, ci limitiamo a riportare la funzione costo derivata rispetto ai pesi (equazione 4).

Exploration vs. Exploitation Spesso quando si affrontano problemi del genere sorge una difficoltà: bilanciare il comportamento dell'algoritmo fra l'esplorazione di nuove mosse casuale in un certo stato, che non ha mai provato prima, oppure essere più *conservatori*, seguendo rigorosamente la propria esperienza rigettando le novità. La soluzione consiste nell'essere tutti e due al momento giusto, infatti il DQN partirà preferendo quasi esclusivamente le nuove esperienze, diventando via via più fedele alla propria politica di comportamento man mano che diventa più attendibile. Per bilanciare quindi fra loro i fattori *exploration* ed *exploitation* nello spazio comportamentale in un determinato stato, l'algoritmo seleziona a in maniera casuale con una probabilità ϵ, oppure segue la politica π con una probabilità $1 - \epsilon$. Ma ϵ partirà con un valore prossimo a 1 è verrà *raffreddato* al valore finale 0.1 verso il milionesimo step.

Experience Replay (ER) Ad ogni passo di tempo t viene archiviata l'esperienza dell'agente costituita dalla tupla $e_t = (s_t, a_t, r_t, s_{t+1})$, che esprime rispettivamente lo stato in cui ha scelto una determinata azione, la ricompensa ricevuta e lo stato di arrivo. L'insieme delle esperienze è D, ed esso accumula esperienze di molteplici episodi fino a un determinato numero N (l'esperimento originario ne accumula un milione). Ogni volta che un *minibatch* fornisce una nuova quantità di esperienze, ne vengono raccolte altrettante nell'insieme D e vengono aggiornate con quelle nuove. In breve, seguono alcuni vantaggi nell'utilizzo dell'ER:

- ogni esperienza può essere usata in molteplici aggiornamenti dei pesi della Q–Network;
- rendere casuale il campionamento di esperienze da aggiornare rompe la correlazione fra eventi consecutivi.

Q–Network Separata L'altra modifica è quella di aggiungere un'ulteriore rete deep per generare i label y_i. Ogni C aggiornamenti la funzione Q viene clonata come $\hat{Q}$, dove $\hat{Q}$ verrà usata per generare etichette y_i per Q per i prossimi C passi. Anche questo è un accorgimento per evitare ciò che accade nel Q–Learning classico dove, un incremento di $Q(s_t, a_t)$ influisce su $Q(s_t, a)$ per qualsiasi valore di a e quindi per tutte le etichette y_i. Così facendo viene quindi evitata la divergenza dalla politica comportamentale π.

$$\nabla_{\Theta_i} L(\Theta_i) = \mathbb{E}_{s,a,r,s'}[(r + \gamma \max_{a'} Q(s', a'; \Theta_i^-))\nabla_{\Theta_i} Q(s, a; \Theta_i)] \tag{4}$$

iii. Implementazione in LUA

Il linguaggio usato da *DeepMind* per l'implementazione dell'algoritmo DQN è `LUA`. Si tratta di un linguaggio di programmazione che punta su efficienza e *leggerezza*, e supporta diversi paradigmi come la programmazione procedurale, orientata agli oggetti, funzionale e programmazione descrittiva dei dati. `LUA` è noto per essere usato in ambiti ludici (*World of Warcraft* e molti giochi per android sono scritti in `LUA`) e, a quanto pare anche *DeepMind*, avendo a che fare con i videogiochi, non si è allontanata troppo dal trend.

Dipendenze Per sperimentare quello di cui si sta parlando, è stata usata una fork del codice originario che permette di vedere anche graficamente sia l'apprendimento che l'esecuzione dei test[1]. In breve, sono state usate le seguenti principali dipendenze:

- *Xitari*, fork di *The Arcade Learning Envoirnment*[2] che permette a chiunque di sviluppare agenti di intelligenza artificiale in grado di interfacciarsi con l'emulatore Atari 2600 *Stella*[3];
- `AleWrap`, interfaccia `LUA` per `Xitari`;
- `LuaJIT`, compilatore *Just in Time* per `LUA`;
- `Torch 7.0`, framework per computazioni di carattere scientifico e ampio supporto ad algoritmi di apprendimento automatico;
- `nn`, libreria potente per fornire un metodo facile e modulare per costruire e allenare reti neurali (molto simile a `Keras` con i suoi modelli sequenziali).

È possibile allenare il proprio agente usando le proprie `CPU` o `GPU` (o una in particolare) tramite i relativi script. Ogni script contiene i seguenti iperparametri (parametri che servono prima del lancio): passi da eseguire, ogni quanti passi eseguire la validazione, valore iniziale di γ ed ϵ, learning rate ed altri parzialmente riportati nella Figura 1. L'implementazione in `LUA` è abbastanza corposa quindi verranno ora riportate le parti ritenute fondamentali.

iii.1 train_agent

Si tratta del file con codice `LUA` che:

- contiene la ricezione e l'elaborazione degli iperparametri visti prima;
- viene creato l'ambiente del gioco selezionato (possibili ricompense e possibili azioni) e viene istanziato l'agente con tutte le sue opzioni;
- a tempo di esecuzione vengono mantenute tutte le informazioni riguardarni il tempo stesso di esecuzione, numero di ricompense, numero di episodi e di step già fatti, e così via;

Ma l'importanza di `train_agent` nella gerarchia dei file sorgente è data dalla presenza del ciclo che itera i passi di apprendimento fino al massimo dichiarato (che di default ricordiamo essere $50mln$). Dato che il ciclo contiene parecchio codice, viene riportata la sua versione riassuntiva nella figura 2.

iii.2 NeuralQLearner

Alla riga 7 in Figura 2 che riporta il ciclo while nel file `train_agent`, è presente il metodo

```
perceive(reward, screen, terminal).
```

Esso appartiene a `NeuralQLearner.lua`. Si tratta dell'implementazione stessa dell'algoritmo DQN, di conseguenza ecco quanto accade qui alla sua istanziazione:

- gli iperparametri vengono usati per inizializzare la rete convoluzionale di cui si parlerà più tardi in II;

[1] DeepMind Atari Q–Learner, *GitHub Repository.*

[2] ALE, *Arcade Platform for General Artificial Intelligence development.*

[3] Stella, *Atari 2600 Emulator.*

Figura 1: *Parte dello script di lancio* `run_cpu`

```
#!/bin/bash

agent="NeuralQLearner"
netfile="\"convnet_atari3\""
update_freq=4                # Aggiornamento ogni 4 frame
actrep=4
discount=0.99                # Valore iniziale gamma
replay_memory=1000000        # Dimensione dell'insieme D (Transition Table)
eps_end=0.1                  # Valore finale di epsilon
eps_endt=replay_memory       # Assestare epsilon la prima volta che D risulta pieno
lr=0.00025                   # Learning rate
state_dim=7056               # Dimensione stato data da 84 per 84 pixel

# Altri iperparametri: valore iniziale di epsilon, range ricompense, ecc...

agent_params="lr="$lr",ep=1,ep_end="$eps_end",ep_endt="$eps_endt",discount="$discount",
    hist_len=4,learn_start="$learn_start",replay_memory="$replay_memory",update_freq="
    $update_freq",n_replay="$n_replay",network="$netfile",preproc="$preproc_net",
    state_dim="$state_dim",minibatch_size=32,rescale_r=1,ncols="$ncols",bufferSize=512,
    valid_size=500,target_q=10000,clip_delta=1,min_reward=-1,max_reward=1"

steps=50000000               # Step totali
save_freq=125000             # Frequenza di salvataggio dei parametri
```

Figura 2: *Ciclo* `while` *fondamentale riportato in maniera riassuntiva nel file* `training_agent`

```
-- Preleva per la prima volta uno stato e la ricompensa
local screen, reward, terminal = game_env:getState()

while step < opt.steps do
    step = step + 1
    -- Avendo lo stato, percepiscilo e scegli l'azione ottimale da eseguire
    local action_index = agent:perceive(reward, screen, terminal)

    if not terminal then
        --[[ Se il gioco non e' finito:
             1. esegui lo step compiendo l'azione appena scelta;
             2. ottieni lo stato e la ricompensa risultante.          ]]
        screen, reward, terminal = game_env:step(game_actions[action_index], true)
    else
        screen, reward, terminal = game_env:newGame()

    --[[ Output di informazioni, attivita' di garbage collector      ]]
    --[[ Esecuzione attivita' di validazione ogni V step             ]]
    --[[ Esecuzione attivita' di salvataggio parametri ogni S step   ]]
end
```

- vengono inizializzati i parametri che regolano il comportamento **exploration vs. exploitation** (cioè ϵ e relativi parametri di raffreddamento durante il processo di apprendimento)
- inizializzazione e raffreddamento del *learning rate* (visto in i);
- il così detto `discount factor`, che in realtà è il valore γ visto in i;
- istanziazione della classe `TransitionTable`, che è in realtà l'implementazione dell'insieme D contenente le

esperienze, visto in i;

```
self.transitions =
        dqn.TransitionTable(args)
```

- e altri parametri relativi al massimo e minimo della ricompensa, ecc...

Riteniamo importante ora elencare i principali metodi di `NeuralQLearner`, che modificano iterativamente i parametri e i valor inizializzati elencati appena sopra:

1. `preprocess(rawstate)`

   Prende in input l'immagine grezza perché poi sia pronta per essere elaborata dalla rete (preprocessamento spiegato in i.1);

2. `getQUpdate(args)`

   Per ogni azione viene ritornato l'aggiornamento del valore Q, ricordiamo che la rete neurale approssima il valore ottimale per ogni azione disponibile. Si tratta di un'iterazione espressa dall'equazione 3;

3. `qLearnMinibatch`

   Dopo aver ottenuto il nuovo valore di Q invocando `getQUpdate(args)`, il metodo aggiorna i pesi della rete neurale tramite il calcolo del gradiente, in base al minibatch di nuove esperienze che poi verranno salvate anche nella `TransitionTable`. Questo è quanto esprime l'equazione 4.

4. `perceive(reward, rawstate, terminal testing, testing_ep)`

   Il metodo `perceive(args)` esprime la principale iterazione e quindi viene riportato nella Figura 3;

5. `eGreedy(state, testing_ep)`

   Dato il valore aggiornato di ϵ, prende un valore casuale che se è minore di ϵ ritorna la prossima azione come casuale, altrimenti invoca il metodo `greedy(state)`;

6. `greedy(state)`

   Dato lo stato, valuta tutte le azioni possibili e ritorna l'azione ritenuta ottimale.

II. Reti Neurali Convoluzionali

Le reti neurali convoluzionali (CNN) sono usate con grande successo per problemi di riconoscimento automatico di pattern bidimensionali come la rivelazione di oggetti, facce e loghi nelle immagini. Le normali reti neurali stratificate con un'architettura Fully Connected (FC), dove ogni neurone di ciascun layer è collegato a tutti i neuroni del layer precedente (neuroni bias esclusi), in generale non scalano bene con l'aumentare delle dimensioni delle immagini. Le CNN sono costituite da neuroni collegati tra loro tramite rami pesati e anche per questa tipologia di rete i parametri allenabili sono: *weight* e *bias*. La fase di allenamento di una rete neurale attraverso *forward/backward propagation* e aggiornamento dei *weight*, vale anche in questo contesto.

Le CNN prendono vantaggio dal fatto che l'input consiste in immagini e quindi vincolano l'architettura in modo più sensibile. A differenza di una normale rete neurale, gli strati di un CNN hanno neuroni disposti in tre dimensioni: larghezza, altezza e profondità. Inoltre, i neuroni di un layer sono connessi solo ad una piccola regione del layer precedente, invece che a tutti i neuroni come in un'architettura FC. Questa è la principale caratteristica che contraddistingue una CNN da una normale rete neurale stratificata con un'architettura fully connected. In figura 4 si può vedere come vengono disposti i neuroni all'interno di una CNN, infatti ciascun layer trasforma un volume 3D di input in un volume 3D di output; quest'ultimo costituisce l'insieme delle attivazioni dei neuroni di tale layer, tramite una determinata funzione di attivazione differenziabile.

Questa tipologia di rete usa principalmente tre idee di base: *local receptive fields*, *shared weights and biases* e *pooling*.

Consideriamo che i pixel dello strato di input siano connessi con quelli dello strato hidden, ma non tutti, bensì solo alcuni di questi. Ogni neurone del primo strato nascosto è connesso ad una regione (ad esempio 5x5) dello strato di input, questa regione dell'immagine dello strato di input è chiamata **local receptive field**

Figura 3: *Il metodo* `perceive(args)` *della classe NeuralQLearner, approssimante la funzione Q con l'ausilio di reti* `CNN`.

```
function nql:perceive(reward, rawstate, terminal, testing, testing_ep)
    --[[ Pre-elaborazione dello stato (sara' null se il gioco e' terminato) ]]
    local state = self:preprocess(rawstate):float()
    local curState

    --[[ Mapping della ricompensa nel range da -1 a 1, poi si prosegue... ]]

    self.transitions:add_recent_state(state, terminal)
    local currentFullState = self.transitions:get_recent()

    --[[ Archivia transizione s, a, r, s' ]]
    if self.lastState and not testing then
        self.transitions:add(self.lastState, self.lastAction, reward,
                             self.lastTerminal, priority)
    end

    if self.numSteps == self.learn_start+1 and not testing then
        self:sample_validation_data()
    end

    curState= self.transitions:get_recent()
    curState = curState:resize(1, unpack(self.input_dims))

    --[[ Esegui selezione di azione ]]
    local actionIndex = 1
    if not terminal then
        actionIndex = self:eGreedy(curState, testing_ep)
    end

    self.transitions:add_recent_action(actionIndex)

    --[[ Esegui aggiornamento Q Learning ]]
    if self.numSteps > self.learn_start and not testing and
        self.numSteps % self.update_freq == 0 then
        for i = 1, self.n_replay do
            self:qLearnMinibatch()
        end
    end

    if not testing then
        self.numSteps = self.numSteps + 1
    end

    self.lastState = state:clone()
    self.lastAction = actionIndex
    self.lastTerminal = terminal

    if self.target_q and self.numSteps % self.target_q == 1 then
        self.target_network = self.network:clone()
    end

    if not terminal then
        return actionIndex
    else return 0 end
end
```

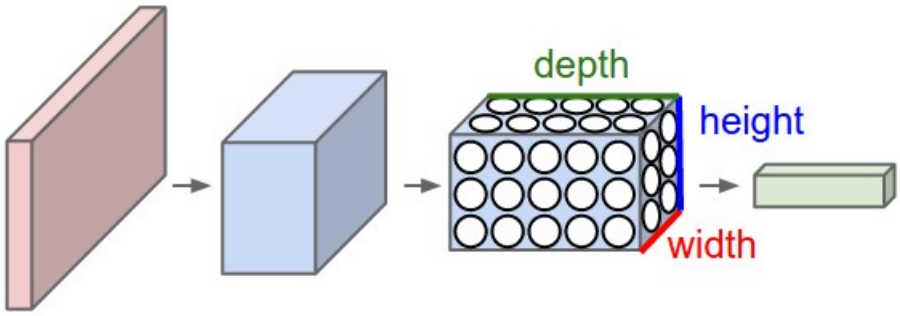

Figura 4: *Convolutional Neural Network*

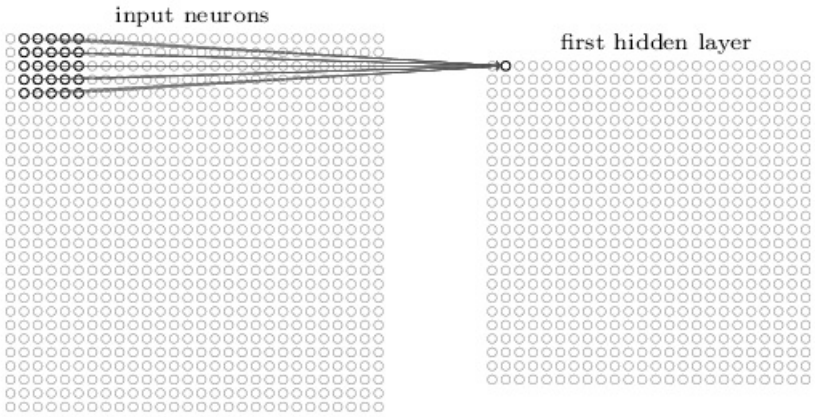

Figura 5: *Local receptive fields*

(figura 5). Ogni connessione apprende un peso e il neurone a cui è associata la connessione apprende un bias totale. In sostanza ogni singolo neurone esegue la convoluzione e la rete è addestrata ad apprendere i pesi e il valore del bias per minimizzare la funzione di costo. Quindi vengono connesse tutte le regioni dello strato di input ai singoli neuroni nel primo strato nascosto, effettuando di volta in volta uno shift. Così facendo se in input avessimo un'immagine 28x28 e le regioni 5x5 otterremo 24x24 neuroni nello strato nascosto.

Dato che i neuroni del primo strato nascosto condividono pesi e bias, questo significa che saranno in grado di riconoscere la stessa feature, solo collocata diversamente nell'immagine di input. Questo rende le CNN molto adattabili all'invarianza di un'immagine ad una traslazione. Per questa ragione, la mappa di connessioni dallo strato di input a quello nascosto è denominata *feature map*, i pesi **shared weights** e i bias **shared bias** perché condivisi. Il vantaggio di condividere i pesi e bias sta nella riduzione dei parametri coinvolti in una rete convoluzionale.

Ovviamente per riconoscere un'immagine sono necessarie più di una feature map, quindi, uno strato convoluzionale completo è fatto da più feature maps. Una CNN può comporsi di più strati di convoluzione collegati in cascata. L'output di ogni strato di convoluzione è un insieme di matrici di convoluzione (ciascuna generata da una mappa di attivazione). L'insieme di queste matrici definisce un nuovo volume di input utilizzato dalle mappe di attivazione dello strato successivo. Più strati convoluzionali possiede una rete e più feature dettagliate essa riesce ad elaborare.

Le CNN usano anche degli strati di **pooling** posizionati subito dopo agli strati convoluzionali, questi hanno la funzione di semplificare l'informazione di output dello strato precedente. Uno strato di pooling suddivide le matrici convoluzionali in regioni e seleziona un unico valore rappresentativo (valore massimo *max-pooling* o valore medio *average pooling*) al fine di ridurre i calcoli degli strati successivi e aumentare la robustezza delle feature rispetto alla posizione spaziale. In sostanza il pooling sottocampiona spazialmente ogni mappa di feature di input, ovviamente il pooling viene applicato ad ogni feature maps separatamente.

L'ultimo strato di una CNN è esattamente uguale ad uno qualsiasi dei layer di una classica rete neurale artificiale con architettura FC: collega quindi tutti i neuroni dallo strato di max-pooling a tutti i neuroni di uscita, utili a riconoscere l'output corrispondente. Questo tipo di layer, a differenza di quanto visto finora nelle reti neurali convoluzionali, non utilizza la proprietà di connettività locale: un FC layer è connesso all'intero volume di input e quindi, come si può immaginare, si avranno moltissime connessioni. L'unico parametro impostabile di questo tipo di layer è il numero di neuroni K che lo costituiscono. Ciò che fa sostanzialmente un FC layer è quello di collegare i suoi K neuroni con tutto il volume di input e di calcolare l'attivazione di ciascuno dei suoi K neuroni. Il suo output sarà infatti un singolo vettore 1 x 1 x K, contenente le attivazioni calcolate. Il fatto che dopo l'utilizzo di un singolo FC layer si passi da un volume di input, organizzato in 3 dimensioni, ad un singolo vettore di output, in una singola dimensione, fa intuire che dopo l'applicazione di un FC layer non si può più utilizzare alcun layer convoluzionale. La funzione principale dei FC layer nell'ambito delle reti neurali convoluzionali è quello di effettuare una sorta di raggruppamento delle informazioni ottenute fino a quel momento, esprimendole con un singolo numero (l'attivazione di uno dei suoi neuroni), il quale servirà nei successivi calcoli per la classificazione finale. Solitamente si utilizza più di un layer FC in serie e l'ultimo di questi avrà il parametro K pari al numero di classi presenti nel dataset. I K valori finali saranno infine date in pasto all'output layer, il quale, tramite una specifica funzione probabilistica, effettuerà la classificazione. Combinando insieme queste tre idee avremmo una rete convoluzionale completa, la quale sarà addestrata tramite la discesa del gradiente e l'algoritmo di backpropagation.

Da un certo punto di vista, possiamo dire che una rete CNN, per la classificazione, è un sistema che si compone di due componenti fondamentali:

- **Componente di riconoscimento ed estrazione automatica delle feature**.
  Il riconoscimento avviene mediante applicazione di banchi paralleli di filtri convoluzionali i cui elementi sono appresi automaticamente mediante algoritmo del gradiente discendente (essi corrispondono ai pesi delle connessioni dei neuroni). Intuitivamente questa componente ha come obiettivo quello di apprendere dei filtri che si attivano in presenza di un qualche specifico tipo di feature in una determinata regione spaziale dell'input.
- **Componente di classificazione**.

L'addestramento di una CNN coinvolge entrambe le componenti in cui i parametri degli elementi costitutivi (p.e. neuroni) sono automaticamente variati al fine di minimizzare una funzione di costo. La capacità di una CNN può variare in base al numero di strati che essa possiede. Raramente si può trovare un solo strato convoluzionale, a meno che la rete in questione non sia estremamente semplice. Di solito una CNN possiede una serie di strati convoluzionali: i primi di questi, partendo dallo strato di ingresso ed andando verso lo strato di uscita, servono per ottenere feature di basso livello, come ad esempio linee orizzontali o verticali, angoli, contorni vari, ecc; più si scende nella rete, andando verso lo strato di uscita, e più le feature diventano di alto livello, ovvero esse rappresentano figure anche piuttosto complesse come dei volti, degli oggetti specifici, una scena, ecc. In sostanza dunque più strati

convoluzionali possiede una rete e più feature dettagliate essa riesce a riconoscere.

i. CNN nel contesto DQN

i.1 Preprocessing

Dopo aver discusso la struttura di una CNN è doveroso delineare il modo in cui l'agente DQN fa uso di questa rete. Prima di imputare i dati nella CNN, viene eseguito un metodo di *preprocessing* con l'obiettivo di ridurre la dimensione dell'immagine di input. Dovendo elaborare ogni frame del gioco, dove ogni pixel ha una dimensione di 210 x 160 con una paletta di 128 colori, diventa molto dispendioso in termini di calcolo e requisiti di memoria. Quindi si applica un passo di *preprocessing* di base volto a ridurre la dimensionalità di input e affrontare alcuni artefatti dell'emulatore *Atari 2600*. Per codificare un singolo frame, viene tenuto il valore massimo di colore per ogni pixel tra il frame da codificare con quello precedente. Ciò permette di rimuovere il cosiddetto *flickering* presente nei giochi Atari, in cui alcuni oggetti appaiono solo in frame pari mentre altri appaiono solo in frame dispari; un artefatto causato dal numero limitato di "nemici" presenti nei giochi Atari che l'emulatore però visualizza contemporaneamente. Fatto questo, viene poi estratto il canale Y, conosciuto come luminanza, dal frame RGB e viene scalato ad una dimensione di 84 x 84.

i.2 Architettura del modello

L'architettura schematizzata della rete è mostrata in figura 6 ed è strutturata come segue. L'input della CNN è un frame che consiste nello snapshot del gioco, con dimensioni 84 x 84 x 4 prodotta dal *preprocessing*. Il primo strato nascosto convoglia 32 filtri di dimensione 8 x 8 con passo 4 dall'immagine di input e viene applicato poi un rectifier (rettificatore) non lineare. Ogni strato nascosto della rete è seguito da uno strato rettificatore che ha come funzione principale quella di incrementare la proprietà di non linearità della funzione di attivazione, accennata in precedenza, senza andare a modificare i receptive field di un layer convoluzionale. Una funzione molto adatta a questo scopo, e per questo è quella utilizzata dai *ReLU* layer, è $f(x) = max(0, x)$. Il secondo strato nascosto convoglia 64 filtri di dimensione 4 x 4 con passo 2 seguito dallo strato ReLU. Questo è seguito da un terzo strato convoluzionale che convoglia 64 filtri di dimensione 3 x 3 con passo 1 seguito anch'esso da uno strato ReLU. L'ultimo strato nascosto della rete è fully-connected e consiste di 512 unità di rectifier. Lo strato di output è uno strato lineare FC con un singolo ouput per ogni azione valida. Il numero di azioni valide variano fra 4 e 18 nei giochi Atarti considerati dallo studio.

i.3 Implementazione

Dopo aver descritto l'architettura della CNN utilizzata per questo studio, analizzeremo l'implementazione della struttura. Come accennato in precedenza, il tutto è stato sviluppato in *Lua* con riferimenti a librerie di IA, ma per lo sviluppo della CNN è indispensabile la libreria *Torch*, un framework scientifico di calcolo con ampio supporto per gli algoritmi di machine learning che predilige le GPU. In figura 7 viene mostrata la parte di codice della funzione *create_network(args)* che implementa i layer convoluzionali con diversi parametri come descritto nella sezione precedente. L'ultima parte della CNN (figura 8) vengono implementati gli strati FC, dove vengono collegati tutti i neuroni di uscita, con la funzione principale di effettuare una sorta di raggruppamento delle informazioni ottenute fino a quel momento esprimendole con un singolo numero (l'attivazione di uno dei suoi neuroni), ovvero, l'azione da compiere.

III. Game Over

L'allenamento dell'agente DQN, spiegato nelle sezioni precedenti e con la stessa struttura della rete, è stato eseguito su 49 giochi *retro* della console *Atari 2600*. In [1] si possono vedere i risultati di tali esperimenti. Quello che sorprende di più è che sono stati raggiunti ottimi

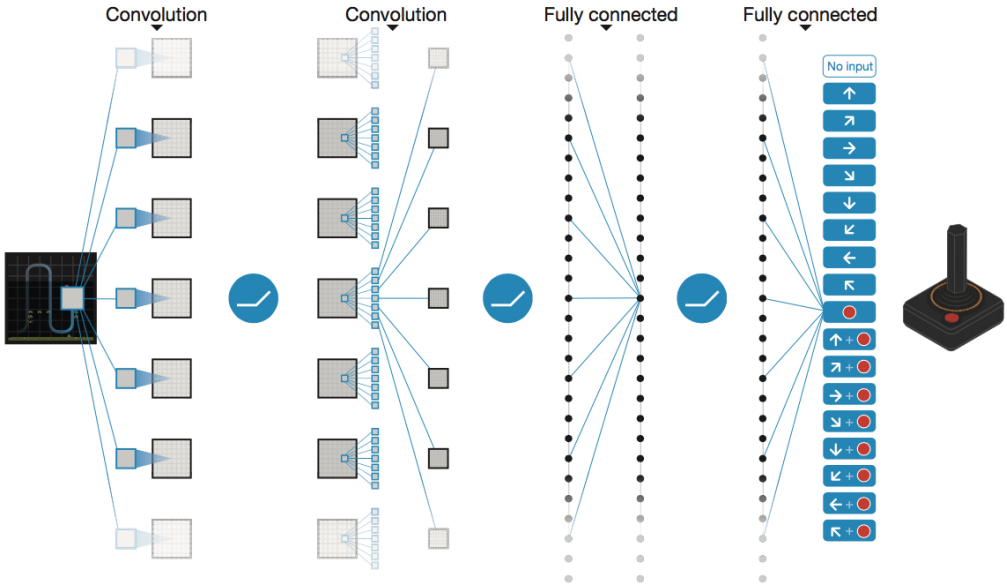

Figura 6: *Architettura della rete neurale deep usata.*

```lua
--[[ Instaziazione del modello sequenziale e aggiunta di un primo strato
     Reshape che rimodella la forma dei dati per i prossimi strati
     convoluzionali. ]]
local net = nn.Sequential()
net:add(nn.Reshape(unpack(args.input_dims)))

--[[ Aggiunta del primo strato convoluzionale usando anche iperparametri ]]
local convLayer = nn.SpatialConvolution

net:add(convLayer(args.hist_len*args.ncols, args.n_units[1],
                  args.filter_size[1], args.filter_size[1],
                  args.filter_stride[1], args.filter_stride[1],1))
net:add(args.nl())

--[[ I restanti strati convoluzionali ]]
for i=1,(#args.n_units-1) do
    net:add(convLayer(args.n_units[i], args.n_units[i+1],
                      args.filter_size[i+1], args.filter_size[i+1],
                      args.filter_stride[i+1], args.filter_stride[i+1]))
    net:add(args.nl())
end
```

Figura 7: *Parte della creazione del modello sequenziale, che riporta l'aggiunta degli strati convoluzionali tramite la potente libreria* `nn` *per* `LUA`

risultati per una serie di giochi che in termine di genere hanno poco in comune: si va dagli sparatutto come *River Raid*, passando per *Boxing* e infine i giochi di guida tredimensionali come *Enduro*.

Secondo alcuni criteri gli autori hanno anche stabilito prestazioni sotto o sopra al livello umano. Purtroppo, mentre per circa la metà dei giochi l'agente supera le prestazioni umane in maniera impressionante, ci sono 20 giochi su

```
--[[ Aggiunta degli strati densamente connessi ]]
net:add(nn.Linear(nel, args.n_hid[1]))
net:add(args.nl())
local last_layer_size = args.n_hid[1]

for i=1,(#args.n_hid-1) do
    -- add Linear layer
    last_layer_size = args.n_hid[i+1]
    net:add(nn.Linear(args.n_hid[i], last_layer_size))
    net:add(args.nl())
end

--[[ Aggiunta dell'ultimo strato dei target per ogni azione ]]
net:add(nn.Linear(last_layer_size, args.n_actions))
```

Figura 8: *Implementazione del layer FC*

49 in cui l'agente non riesce a fare meglio. Si vuole provare a individuare il fattore che rende diverse (nei risultati) queste due categorie di giochi, il sospetto è che per come è stato progettato l'algoritmo DQN, esso capti ricompense di alcuni giochi in un modo che non gli permette di imparare comportamenti ottimali.

i. Alcune prove

Pur non disponendo neanche lontanamente delle risorse di calcolo a disposizione di *DeepMind*, l'**intenzione** iniziale era quella di accompagnare lo studio dei risultati non soddisfacenti con alcune prove pratiche, per capire meglio l'ambiente di cui stiamo parlando. Sono state fatte alcune prove su due giochi selezionati da diverse categorie: *Breakout* e *Centipede*. Nell'articolo originale, il primo supera il miglior giocatore umano in maniera che si può definire fuori scala, di ben 1327%, mentre il secondo non ha raggiunto risultati soddisfacenti, fermandosi al 62%.

Gli step totali eseguiti da *DeepMind* sono $50mln$, che su un PC normale equivale ad aspettare in termini di settimane (o poco più di un mese). In un primo tentativo volevamo vedere cosa sarebbe accaduto allenando la rete per soli 700.000 step, scoprendo che circa 12 ore (24 in totale sia per *Centipede* che *Breakout*) non bastano neanche lontanamente per far emergere qualche comportamento sensato da parte dell'agente. Il punteggio di *Breakout* era di soli 55 punti (con l'agente sempre fermo alla sinistra dello schermo con qualche tremolio verso destra) mentre la rete allenata per *Centipede* accumulava soli 1811 punti. In un secondo tentativo di $1,25mln$ di step, ipotizzando che il riempimento della `TransitionTable` migliorasse in qualche modo la situazione, per solo *Centipede* è stato raggiunto il non molto brillante punteggio di 1911.

È chiaro che per computazioni del genere in maniera efficiente non basta il computer di casa. Bisogna specificare però che è stata usata la `CPU` che è un `i5 4690K`, dato che dopo aver provato due versioni di `Ubuntu` non risultava essere alcun driver `CUDA GTX980` compatibile al giorno d'oggi anche con l'implementazione del DQN. Per quanto riguarda il lato pratico quindi, è tutto quello che è stato possibile realizzare.

IV. Prestazioni scadenti: cause individuate

Prima di tutto era necessario capire l'intero ramo dell'apprendimento per rinforzo, il QL e infine le DQN, fatto ciò è possibile passare allo studio dei risultati presenti in [1]. L'attenzione era inizialmente focalizzata sul risultato di *Centipede*, ma per capire il perché delle sue prestazioni non brillanti era anche necessario fare il confronto con altri giochi. Guardando i *gameplay* della maggior parte del parco titoli,

e consultando i relativi manuali per capire la natura delle ricompense (*quante* e *quando*)[4] in relazione alle possibili osservazioni, è risultato chiaro che alcuni giochi in realtà si assomigliano molto per costruzione. Si stava cercando un pattern che distinguesse i giochi in cui l'algoritmo si comportasse bene o male, ma sono stati individuati almeno tre motivi per cui un agente DQN può risultare sotto le prestazioni umane. È necessario ricordare prima di procedere, che la struttura dei giochi non è stata cambiata in alcun modo tranne per il sistema delle ricompense. Qualsiasi ricompensa negativa è stata mappata su -1, su 1 quelle positive e su 0 quelle nulle. A partire da [1] non è chiaro se la fine della partita venga considerata come ricompensa negativa, in seguito ad alcune ricerche nel codice sorgente si nota come in realtà l'agente non venga punito in tal caso:

```
self.config = {
...
gameOverReward = 0,
...
}
```

dove la variabile `gameOverReward` non compare più in alcun assegnamento.

i. Pianificazione a lungo termine

Il punto debole più ovvio di un agente DQN è la pianificazione a lungo termine, in mancanza di ricompense intermedie. Si può facilmente notare nella Figura 9 come, nel gioco *Montezuma's Revenge*, per arrivare a prendere la chiave e quindi ricevere la ricompensa, sia necessario eseguire una lunga catena di azioni diverse e osservazioni senza feedback alcuno. Pensiamo che *Private Eye*, ricalcando le stesse modalità di gioco di *Montezuma's Revenge*, abbia per lo stesso motivo raggiunto un risultato quasi nullo. Inoltre, per giocare è necessario muoversi contemporanemante su più livelli, logicamente collegati. Non vediamo alcun meccanismo in grado di gestire scelte del genere. Almeno per quanto riguarda la singola chiave, si potrebbe

[4] Atari Age, *Atari 2600 Rarity Guide.*

Figura 9: *L'eroe deve arrivare alla chiave eseguendo molteplici azioni ed evitando il teschio.*

ragionare su ricompense intermedie. Ma ricordiamo l'obiettivo è quello di ottenere un algoritmo in grado di apprendere *end-to-end*, di carattere generale, senza il bisogno di modifiche per task specifiche.

ii. Scelte di carattere tattico

Ci risulta che un'altro punto debole dell'algoritmo proposto da *DeepMind* sia il *fiuto tattico*, che percepiamo come l'abilità di intuire la possibilità di uno scenario sfavorevole e reagire di conseguenza. Non c'è un meccanismo in grado incoraggiare tale comportamento, per quanto riguarda *Pac-Man* è vero che l'agente viene punito se va contro il fantasma, ma questi cambiano spesso direzione e (casualmente o no) creano delle zone trappola. Per esempio, si osservi la Figura 10a: una sfavorevole situazione evitabile da prima facendo azioni migliori. Purtroppo, anche in questo caso l'agente DQN otterrebbe tutte le ricompense che gli spettano, senza una previsione iniziale della direzione del fantasma. Un essere umano rimane più bravo a indirizzare *Pac-Man* in una parte del labirinto in cui non ci sono punti da guadagnare, per poi tornare nella parte interessante del labirinto sotto una condizione più favorevole (Figura 10b).

Controesempio Per quanto riguarda il titolo *Breakout*, in [1] viene riportato un caso in cui l'algoritmo individua la furbizia di creare un unico foro nel muro per mandarci la palla dietro (Figura 11). In un primo momento potrebbe sembrare che questo sia in contrasto con le conclusioni fatte per quato riguarda *Ms. Pac-Man*. Tuttavia, pensiamo che in *Breakout*

(a) *Situazione sfavorevole, ma ancora riparabile, guidata per la maggiore dalla voglia di ottenere ricompense istantanee.*

(b) *Esempio di scelta sensata, quella di andare a raccogliere punti in un'altra porzione della mappa mentre i fantasmi sono impegnati altrove.*

la valutazione dell'ottimalità della sequenza di azioni non cambi molto a causa di eventi imprevisti. È questione di ambiente: in *Breakout* non ci sono insidie improvvise (o di qualsiasi altro tipo). A parte questo, il gioco di *Breakout* (così come *Pong*) premiano le *skill* di reattività che vengono perfettamente ricompensate in maniera istantanea. *Pac-Man* (così come *Alien*) non è solo questo.

iii. Ricompense Ambigue ed elevata Complessità

Un'altro motivo che ci è parso plausibile per la mancanza di prestazioni ottime in alcuni giochi, è perché non sono "semplici" quanto *Breakout* e simili sparatutto (*Space Invader, Deamon Attack*), dove bisogna andare a destra e a sinistra nel momento giusto; si intendono anche i giochi come *Robotank* che sembrano diversi e addirittura in 3D, dove in realtà si tratta sempre di mirare verso destra o sinistra e sparare al momento giusto.

Per dirla in altre parole, ci sono giochi come *Centipede* che aggiungono un grado di difficoltà in più: i nemici sono numerosi e con comportamenti diversi, non si muovono solo in orizzontale ma anche in verticale circondando il giocatore da tutte le direzioni possibili. Come già detto, ci sono tipi di nemici con comportamenti diversi la cui uccisione comporta a un punteggio diverso: sarebbe saggio uccidere il più in fretta possibile le parti del centipede che altrimenti verrebbero a minacciare il giocatore ai lati o dietro le spalle. Tuttavia, ricordiamo che l'algoritmo DQN mappa tutte le ricompense positive al valore 1. Nelle situazioni più calde alle quali l'algoritmo viene sottoposto (visibile nella Figura 12), esso fa fatica a muoversi in tutte le quattro direzioni continuando a sparare. In più è stato notato che appena dopo la morte del giocatore (punizione -1) il gioco si ferma e assegna una ricompensa $+1$ per ogni fungo (quadratini sullo schermo). Questo potrebbe, come minimo, insegnare all'algoritmo che tutto sommato morire non è poi talmente tragico.

V. Sessioni di apprendimento

In questa sezione verranno riportate le esperienze ottenute in seguito a una serie di prove di apprendimento in due giochi: *Breakout* e *Centipede*. Inizialmente si era pensato di fare circa 50 ore di apprendimento per ognuno dei due giochi, questo significa circa 18 milioni di step al posto dei 50 milioni al contrario eseguiti eseguiti da *DeepMind*. Per valutare le prestazioni degli agenti verrà usata la stessa formula 5 presente in [1].

$$100(dqn - r)(human - r) \tag{5}$$

Dove *dqn* sono i punteggi medi dell'agente DQN su più prove, *human* dell'essere umano, e r i punteggi ottenuti giocando scegliendo azioni casuali. Tutti questi dati sono disponibili, per ogni gioco, sempre in [1], l'unica cosa che verrà cambiata sarà il valore *dqn*, stavolta ottenuto in seguito alle nostre prove.

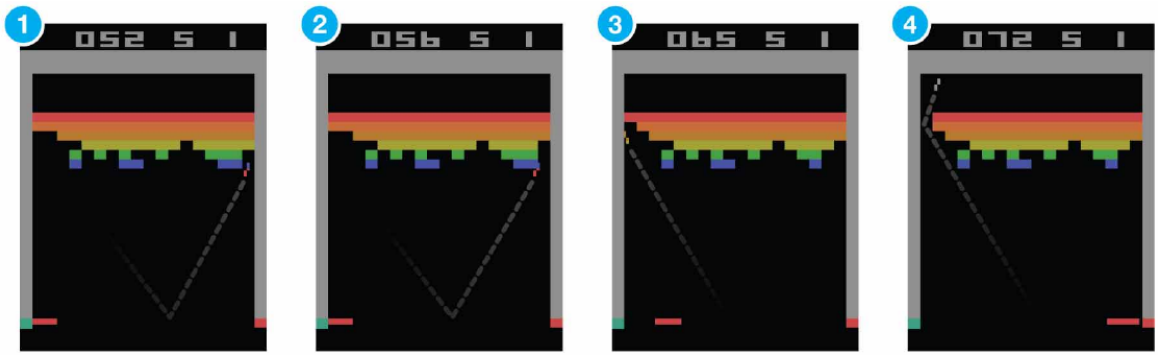

Figura 11: *Dopo un certo numero di ore di apprendimento l'algoritmo DQN è in grado di individuare situazioni in cui una sola azione porta numerose ricompense.*

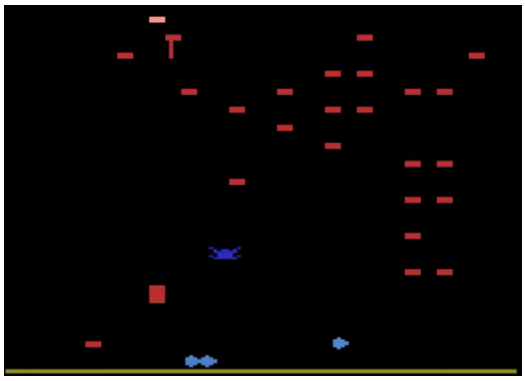

Figura 12: *Situazione (neanche delle più difficili) in cui l'elfo si ritrova con pezzi di centipede e altri nemici che si muovono a grande velocità intorno al protagonista.*

Tabella 1: *Esempio di sistema di ricompense in un gioco Atari 2600: Centipede. Rispetto ad altri giochi è piuttosto esteso, il suo mapping a* -1 *o* 1 *porta troppe semplificazioni, soprattutto se si prende in considerazione il momento in cui vengono date.*

Tipo	Quantità Punti
Centipede parte del corpo	10
Centipede testa	100
Ragno (lunga distanza)	300
Ragno (media distanza)	600
Ragno (stretta prossimità)	900
Pulce	200
Scorpione	1000
Funghi eliminati	1
2 Funghi (danneggiati o intatti alla fine)	5
Funghi avvelenati eliminati	1
2 Funghi avvelenati (danneggiati o intatti alla fine)	5
Nuova Bacchetta Magica	10.000

i. *Breakout*

i.1 Rompere il muro

i.2 Situazione di stallo

VI. Evoluzione dell'ambito RL

La pubblicazione dell'algoritmo DQN proposto in [1] risale al 2015, la sua implementazione molto probabilmente ancora prima (dovrebbe essere 2014). In ambito *Machine Learning* significa che in realtà è un algoritmo vecchio: da allora non solo sono state proposte modifiche del DQN ma anche alternative. La ricerca di *DeepMind* stessa si è spostata su altri metodi. Si andrà ora a vedere brevemente, senza entrare

troppo nel merito, le modifiche apportate al DQN in diversi articoli.

La stessa *DeepMind* se ne accorse in un secondo momento che DQN sovrastima in molti casi alcuni valori di Q (cioè il valore delle azioni da eseguire). È chiaro che questo introduce errori che possono sviare l'agente dall'apprendimento di un comportamento tendende all'ideale. Gli autori della *Double DQN* [4] hanno proposto quindi il seguente principio: anziché prendere il massimo valore di Q per ogni step viene prima usata una rete deep per scegliere l'azione, poi una seconda per generare il valore Q di tale azione. In pratica è stato divisa la scelta di *a* dalla generazioni dei valori di Q, e questo ha permesso la riduzione della sovrastima. Come si può vedere in [4], l'algoritmo DDQN ha migliorato parecchio i risultati ottenuti, ma solo nei giochi in cui aveva già ottenuto buoni score. Persistono ancora i problemi trattati in IV.

Sono state proposte modifiche all'*Experience Replay* in [2], dove viene suggerito che non tutte le transizioni sono uguali. La tecnica `Prioritized Replay` prevede che vengano campionate con precedenza le esperienze in base a diversi criteri. Uno di questi, è guardare con più attenzione le transizioni che hanno introdotto un errore maggiore sui valori di Q, e ciò significa che portano con sè più informazioni. Tale tecnica introduce migliorie per 41 giochi sui 49 presenti in [1]. Sono state proposte (anche se, senza test esaustivi su tutti i giochi) particolari tecniche per il raffreddamento di ϵ; sempre da *DeepMind* è stato proposto un algoritmo in grado di apprendere in ambienti con controlli continui [3].

Esistono altri metodi per attaccare i problemi RL, come il metodo *Policy Gradient*[5], che tenta di apprendere funzioni che mappano direttamente le osservazioni ad azioni, senza alcun valore intermedio del trovarsi in un certo stato ed eseguire una certa azione *a* come nel caso di DQN. Un'ultimo algoritmo sempre di *DeepMind* che si vuole qui riportare, l'*Asynchronous Advantage Actor-Critic*[5] (A3C), ha semplicemente reso obsoleti altri algoritmi già usciti. I pilastri di funzionamento su cui si basa sono i seguenti:

- **Asincronia**, ci sono diversi thread con diverse istanze dell'ambiente che apprendono, e che hanno un proprio insieme di parametri all'interno di un'unica grande rete;
- **Actor Critic**, combinazione di entrambi i metodi *Policy Gradient* e *Q—Learning*, ovvero la valutazione di quanto sia bello trovarsi in un certo stato e una distribuzione di probabilità sull'insieme delle azioni, entrambe le cose implementate come strati densamente connessi sopra l'architettura generale;
- **Advantage**, si è in grado di generare un stima del *vantaggio*, che permette all'algoritmo di migliorarsi là dove le previsioni della rete corrispondono meno al vero.

L'algoritmo è stato utilizzato per apprendere il gioco al primo *Doom*. A3C stesso ha parecchie versioni e modifiche, incluso l'utilizzo di reti deep ricorsive, e viene spesso usato per migliorare ulteriormente i risultati in vari ambienti[6]. Ad ogni modo, mentre si possono trovare algoritmi che apportano miglioramenti all'apprendimento in giochi come *Pac-Man*, rispetto al DQN, non ci risulta che esistano algoritmi in grado di stravolgere gli esiti all'interno di *Montezuma's Revenge*.

VII. Conclusioni

Per riuscire ad argomentare nel miglior modo possibile i risultati poco soddisfacenti di un sottoinsieme del parco titoli *Atari 2600*, è stato prima studiato l'algoritmo *Q—Learning*, poi sono state individuate le aggiunte fatte da *DeepMind* per la realizzazione dell'algoritmo *Deep Q—Network*. Dopo aver appurato in che modo è stata realizzata la relativa implementazione in `LUA`, è stato analizzato il contributo delle reti neurali convoluzionali. Per aggiungere valore all'esperienza, era nel nostro intento riuscire ad allenare per davvero degli agenti

[5]Deep Reinforcement Learning: Pong from Pixels, *GitHub Karpathy Blog.*

[6]Open AI Gym, *Atari Environment*.

DQN, ma dopo alcune prove è stato scoperto che per un processo di apprendimento un minimo soddisfacente servono capacità di calcolo non indifferenti e tempo di calcolo misurabile in settimane. Alcuni risultati (seppur non soddisfacenti) sono comunque stati riportati.

In seguito a un'attenta analisi di *gameplay* e regole che stanno alla base di diversi giochi *Atari 2600*, e degli studi precedentemente fatti, sono stati forniti degli argomenti che classificano i giochi nei tre seguenti gruppi di difficoltà: giochi con pianificazione a lungo termine, giochi con parecchie scelte a livello tattico e giochi che semplicemente riteniamo più difficili degli altri.

Infine, dato il tipico rapido invecchiamento di alcune pubblicazioni in ambito *Machine Learning*, è stato dato uno sguardo all'evoluzione dell'argomento trattato.

Riferimenti bibliografici

[1] Volodymyr Mnih, Koray Kavukcuoglu and David Silver, *Human-level control through deep reinforcement learning*, 2015.

[2] Tom Schaul, John Quan, Ioannis Antonoglou and David Silver, *Prioritized Experience Replay*, 2016.

[3] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver and Daan Wierstra *Continuous Control with Deep Reinforcement Learning*, 2016.

[4] Hado van Hasselt, Arthur Guez and David Silver, *Deep Reinforcement Learning with Double Q-learning*, 2015.

[5] DeepMind *Asynchronous Methods for Deep Reinforcement Learning*, 2016.