UNIVERSITÀ DEGLI STUDI DI PERUGIA

Corso di Laurea in Informatica

Tesi di Laurea Triennale

Applicazione del Reinforcement Learning a Tetris

Relatore
Prof. Alfredo Milani

Correlatore:
Dott. Giulio Biondi

Laureando
Mattia Polticchia

Luglio 2020

Ai miei amici,
alla mia famiglia
e alle altre persone
che mi hanno supportato
Grazie.

Indice

Capitolo 1

Introduzione

1.1 Intelligenza artificiale

L'intelligenza artificiale sta sempre più prendendo piede nel mondo di tutti i giorni. Per dare una definizione di intelligenza artificiale si deve partire dall'idea di "intelligente". Alcune delle caratteristiche di intelligenza si possono riscontrare in questi comportamenti:

- Nel modo di interagire con gli ambienti che lo circondano
- Nell'abilità di raggiungere un obiettivo
- Nella sua capacità di adattarsi all'ambiente in cui si trova e di adattarsi ai cambi di obiettivo.

Unendo tutte queste particolarità abbiamo un punto di arrivo in questa definizione: *"L'intelligenza misura l'abilità di un AI di raggiungere obiettivi in un ampio spettro di ambienti"*[1]

L'intelligenza artificiale ha trovato spazio tra le discipline accademiche già dal 1955 dove aveva ottenuto grande successo, seguito poi da sfiducia e mancanza di interesse per poi ritrovare consenso e successo. Il successo è dovuto soprattutto

all'utilizzo di approcci nuovi alla materia. Tra questi troviamo metodi statistici e di intelligenza computazionale.[17] Alla base della ricerca sono stati messi dei problemi di ragionamento, rappresentazione della conoscenza, pianificazione, apprendimento, comprensione del linguaggio e la capacità di percepire e manipolare oggetti.

1.2 Machine Learning

Il problema dell'apprendimento viene studiato nel campo del Machine Learning, con l' obiettivo di studiare metodi ed algoritmi per permettere alle intelligenze artificiali di apprendere e milgiorare con l'esperienza. Questo avviene tramite la costruzione di un modello matematico basato su dati di addestramento così da permettere di effettuare delle previsioni o di prendere decisioni alle intelligenze artificiali. Uno dei metodi di studio più legati al machine learning è la statistica computazionale con cui si riesce a effettuare predizioni attraverso i computer.

Nel dettaglio il machine learning studia come le Intelligenze Artificiali possano eseguire dei compiti senza essere programmate per farli. Il processo viene arricchito fornendo dei dati da cui possano apprendere determinati comportamenti. Gli approcci di sviluppo sono molteplici ma, per lavori semplici si possono programmare direttamente le AI per permettergli di capire e risolverli. Per lavori più complessi, invece, si può utilizzare un approccio più libero e creare algoritmi appositamente per risolverli o lasciare che si crei il proprio algoritmo risolutivo.[20]

Le tecniche risolutive per questi metodi di sviluppo sono divisibili in tre categorie principali[17]:

Apprendimento Supervisionato : la macchina deve apprendere una funzione che associ dei dati in input al corrispettivo output. L'apprendimento avviene fornendo alla macchina delle coppie input-output atteso di esempio, che saranno utilizzate per creare un modello.

Apprendimento Non Supervisionato : la macchina deve individuare pattern non evidenti all'interno di insiemi di dati non etichettati. Esempi di apprendimento non supervisionato includono *Cluster Analysis*, *Anomaly Detection* e *Latent Variable Models*.

Apprendimento per Rinforzo : Nell'Apprendimento per Rinforzo un agente artificiale interagisce con un ambiente dove deve raggiungere degli obiettivi. L'agente percepisce lo stato dell'ambiente e decide quale azione intraprendere sulla base dell'esperienza precedente, per massimizzare una funzione detta *reward function*.

Un altro metodo per catalogare le tipologie di Intelligenze Artificiali è quello basato sul tipo di problema che risolvono o precisamente sul tipo di output che restituiscono[20]:

- Classificazione : Il compito dell'AI è quello di creare un modello che classifichi gli input.
- Regressione : Si utilizzano le AI per effettuare previsioni su dati dati in input inerenti a valori già noti nel tempo passato.
- Clustering : L'AI è incaricata di analizzare l'input e di dividerlo in gruppi da lei creati.

1.3 Applicazioni

Una pietra miliare in questo campo è AlpaGo che con anni di ricerca è riuscito a battere un campione a livello mondiale in una partita di GO. Lo sviluppo effettuato è stato fatto utilizzando una combinazione tra machine learning e tecniche di ricerca su alberi per poi passare ad una fase di apprendimento sul gioco sia osservando partite fatte da umani sia giocando partite simulate. Nella prima fase di

apprendimento hanno utilizzato tecniche di apprendimento supervisionato basato sul delle partite tra umani fornendogli un dataset di circa trenta milioni di mosse. Dopo che ha raggiunto una certa bravura nella previsione di mosse è stato utilizzato l'apprendimento per rinforzo lasciandola giocare contro altre istanze di se stessa per fissare quanto appreso e dargli un riscontro.[24]

Capitolo 2

Reinforcement Learning

2.1 Storia

In passato l'apprendimento per rinforzo si vedeva diviso in due branche non troppo coese. La prima cercava di simulare l'apprendimento umano e animale tramite tecniche *try and error* mentre l'altra cercava di trovare il comportamento e la soluzione ottima per dei problemi. Per molto tempo prima del 1980 queste due branche si sono concentrate sul metodo risolutivo del problema e non sull'effettivo apprendimento.

I primi utilizzi in letteratura dei termini "Reinforcement" e "Reinforcement Learning" risalgono agli anni '60 [23][11][12], anche se le basi per l'apprendimento "trial and error" si possono attribuire a Farley e Clark, i quali nel 1954 progettano una rete neurale disegnata per apprendere dal processo di *try and error*. Allo stesso tempo, Minsky descrive dei metodi computazionali di Reinforcement Learning nella sua tesi di dottorato; in seguito, nel 1961 lo stesso Minsky pubblica un paper dal titolo *"Steps Toward Artificial Intelligence"* [14], in cui discute le modalità con cui fornire ad un'agente artificiale feedback sulle sue decisioni; egli formalizza il *credit assignment problem*, in cui si esamina come distribuire il punteggio positivo dovuto ad un successo tra tutte le azioni che hanno contribuito al successo stesso.

Donald Michie, tra il 1961 ed il 1963, progetta un semplice sistema di apprendimento con approccio *trial and error* per tic-tac-toe chiamato MENACE (*Matchbox Educable Naughts and Cross Engine*) [13], consistente in una serie di scatole di fiammiferi che rappresentano ognuna una possibile serie di mosse; in ognuna delle scatolette le singole mosse sono rappresentate come perline colorate. Alla fine di ogni partita le perline sono aggiunte o rimosse per punire o rinforzare le decisioni di MENACE. Nel 1968 lo stesso Michie, insieme a Chambers, teorizza un sistema chiamato GLEE (*Game Learning Expectimaxing Engine*), applicato anch'esso al tic-tac-toe; in seguito è sviluppato anche il controller BOXES, a cui è assegnato il compito di imparare a tenere bilanciato un palo in un carrello in un percorso. Il meccanismo di apprendimento è legato ad un singolo riscontro, positivo o negativo, che dipende dall'esito della corsa, che può essere portata a termine o interrotta perché il palo è caduto. Questo è tuttora uno degli esempi di base utilizzati nell'insegnamento dell'apprendimento per rinforzo.

Parallelamente al *try and error* è approcciata e sviluppata l'idea della differenza temporale. Questo approccio è ispirato ai metodi di apprendimento animali, e più precisamente si ispira al concetto di *rinforzo secondario*, che descrive stimoli capaci di rinforzare un comportamento dopo essere stati associati a stimoli primari, come cibo e paura. Minsky nel 1954 intuisce la possibilità di applicare teorie definite nell'ambito della psicologia all'intelligenza artificiale; di seguito Arthur Samuel, nel 1959, sulla scia di questa intuizione propone l'implementazione di un metodo di apprendimento che includesse anche la differenza temporale. In risposta, Minsky discute l'operato di Samuel in *Steps Toward Artificial Intelligence* [14], considerando valida l'intuizione e confermando il collegamento con il regno animale.

Nel 1981, alla luce dei progressi precedenti, è teorizzata da Barto, Sutton e Anderson un'architettura che tiene conto della *temporal-difference* e basata sull'approccio *try and error*, nota come *Actor-Critic Architecture*. Tale architettura è inizialmente applicata al problema del bilanciamento di Michie nel 1983 [2], ed è

successivamente discussa ed estesa da Sutton, che propone di integrare reti neurali addestrate tramite backpropagation.

Infine, nel 1989, Watkins con lo sviluppo del Q-learning raggiunge un nuovo punto di svolta, estendendo ed integrando quanto discusso in precedenza. Questo algoritmo di reinforcement learning, che costituisce una delle basi per tutte le ricerche attuali, è discusso in dettaglio nel paragrafo 2.2.1.

2.2 Reinforcement Learning

L'apprendimento per rinforzo può essere utilizzato in ambienti *non deterministici*, cioè ambienti in cui una determinata azione, eseguita in diversi contesti, non porterà allo stesso stato e/o allo stesso risultato. Un agente ha il compito di scegliere la prossima azione da eseguire in base al proprio stato e alla percezione dell'ambiente circostante, che può essere limitata oppure completa. L'apprendimento per rinforzo si basa sulla libertà dell'agente di interagire con l'ambiente ed adattare i propri comportamenti in base all'esperienza, impararando una strategia di soluzione dei problemi non precedentemente codificata in comportamenti prefissati [21].

Nella fase di esplorazione, l'ambiente ha il compito di restituire dei punteggi in base alle azioni che l'agente ha compiuto, associando valori positivi ad azioni che producono buoni risultati e negativi ad azioni che producono errori. Questo permette all'agente di produrre associazioni azione-risultato, che andranno poi ad influire sulle scelte future.

Criteri di Ottimalità

L'agente può intraprendere delle scelte di azioni da fare, questa scelta è formalizzata come segue:

$$\pi : A \times S \rightarrow [0,1]$$

$$\pi(a, s) = Pr(a_t = a | s_t = s)$$

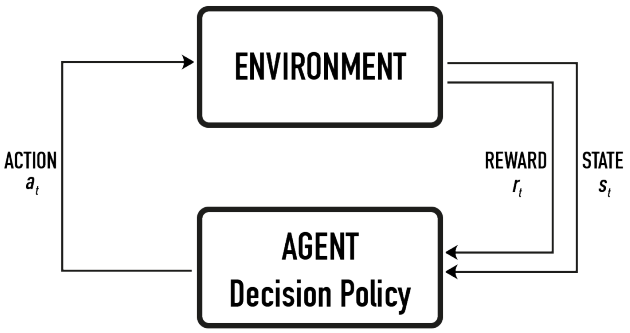
π rappresenta la policy di scelta, A l'insieme delle azioni e S l'insieme degli stati. Come risultato della valutazione restituisce la probabilità di scegliere l'azione a nello stato s.

La funzione di valutazione viene definita come il valore atteso dalla policy allo stato s; nel concreto, la funzione va a valutare quanto sia vantaggioso essere nello stato s.

$$V_\pi(s) = E[R] = E[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s]$$

R è definito come la somma dei valori ottenuti dall'ambiente r a cui viene applicato lo sconto γ.

Figura 2.1: Raffigurazione schematizzata del rapporto tra agente e ambiente



2.2.1 Q-learning

Il Q-learning è un algoritmo di apprendimento per rinforzo che permette all'agente di imparare una policy di scelta che determini quale azione intraprendere in base agli stati in cui esso si trova. Per l'apprendimento non necessita di modello dell'ambiente e riesce a gestire i vari cambi dell'ambiente senza bisogno di modifiche. Questo algoritmo trova una policy ottimale volta a massimizzare il valore del reward totale rispetto ad ogni step e i suoi successivi, partendo dallo stato d'osservazione. Il Q-Learning riesce anche ad ottenere una politica di selezione dell'azione dando un infinito tempo di esplorazione e una policy di scelta parzialmente casuale.

Come nella metodologia teorica del Reinforcement learning, il Q-Learning associa agli stati dei pesi a partire da uno stato chiamato Δt. Il valore previsto negli stati successivi si definisce come $\gamma^{\Delta t}$ dove γ è il fattore di sconto. Questo fattore, compreso tra zero e uno ($0 \leq \gamma \leq 1$), è utile per valutare se c'è della differenza tra il reward attuale e quello previsto. Un'altra chiave di lettura lo rende utile per capire lo stato di successo per ogni step in che viene valutato. Grazie ad esso l'algoritmo

ha modo di capire la qualità dell'andamento tramite la relazione stato-azione

$$Q : S \times A \to \mathbb{R}$$

Q solitamente viene inizializzato lasciando l'algoritmo agire per un determinato numero di step superati i quali avremo che per ogni step t l'agente selezionerà un azione a_t, ne osserverà il reward r_t entrerà nel nuovo stato $s_t + 1$ ed aggiornerà Q. Quanto precedentemente visto può essere così formalizzato

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{vecchio valore}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{tasso di apprendimento}} \times \left(\overbrace{\underbrace{r_t}_{\text{ricompensa}} + \underbrace{\gamma}_{\text{fattore di sconto}} \underbrace{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})}_{\text{valore futuro massimo}}}^{\text{valore appreso}} - \underbrace{Q(s_t, a_t)}_{\text{vecchio valore}} \right)$$

L'episiodio terminerà quando s_t sarà uno stato finale e al valore di $Q(s_f, a)$ verrà attribuito il valore di r allo stato s_f.

Parametri di Tuning

I parametri di tuning (o Iperparametri) sono valori utilizzati per controllare il processo di apprendimento. Si suddividono in due tipologie:

- Del modello:

  Non hanno la possibilità di interferire con il training o il learning perché sono dichiarati all'istanziamento del modello.

- Dell'algoritmo :

  Non influiscono sul comportamento del modello ma influiscono sulla velocità e la qualità del processo di apprendimento.

Learning Rate (step size) : determina dopo quanti step le precedenti informazioni vengono aggiornate. Un valore 0 impedirà all'agente di imparare mentre 1 farà considerare all'agente solo le ultime informazioni. In ambienti deterministici $\alpha_t = 1$ è ottimo, mentre quando l'ambiente è stocastico è buona norma che $\alpha_t = 0.1$ per evitare che l'algoritmo converga e decresca a zero.

Fattore di sconto (gamma) : Il fattore di sconto influisce l'importanza dei reward futuri. Il valore a 0 renderà l'agente interessato ai reward immediati mentre ad 1 lo renderà goloso per i risultati a lungo termine. Con γ vicino ad uno potrebbe verificarsi che l'algoritmo tenda a propagare errori di valutazione con modelli basati su reti neurali. Una soluzione a questo problema si ha iniziando con un fattore di sconto basso e aumentarlo gradualmente.
Parliamo di applicazione dello sconto come dell'operazione di moltiplicazione tra γ e il punteggio ricevuto dall'ambiente.

Initial conditions Q_o (learning starts) : Anche se il Q-learning è eseguito in maniera ricorsiva ha comunque bisogno di uno stato iniziale per la valutazione. Dei valori iniziali ottimi possono facilitare l'esplorazione per poi riottenere dei valori inerenti al contesto grazie al primo r dall'ambiente.

2.2.2 Varianti del Q-Learning

Deep Q-Learning (DQN)

Il Deep Q-learning deve questo nome all'utilizzo di una rete neurale convoluzionale profonda costituita da più livelli di nodi interconnessi tra loro, permettendo così una rappresentazione della policy Q più precisa. Questo approccio permette di riproporre l'abilità degli animali chiamata *Experienced Replay*. Per un certo numero di azioni, ne viene selezionata una casualmente e la fa effettuare all'agente. Passato questo numero di azioni, utilizza il Q-learning per aggiornare la politica di scelta in base ai riscontri ottenuti.[15] Questo approccio evita che gli aggiornamenti effettuati dal Q-learning stagnino su un solo sottoinsieme di azioni ma che si tenga conto di tutte le possibili scelte.

Double Q-Learning

Il Double Q-Learning cerca di risolvere il problema della propagazione dell'errore nelle valutazioni di Q che spesso può rallentare l'apprendimento. Non viene più fatta una valutazione Q ma ne vengono fatte due, Q^A e Q^B, e il calcolo del fattore di sconto viene fatto utilizzando una differente politica.[6] Possiamo formalizzare le Q partendo dalla formula vista in precedenza:

$$Q^A_{t+1}(s_t, a_t) = Q^A_t(s_t, a_t) + \alpha Q(s_t, a_t) \cdot (r_t + \gamma Q^B_t \cdot (s_{t+1}, \max_a Q^A_t(s_t + 1, a) - Q^{A_t}(s_t, a_t)),$$

$$Q^B_{t+1}(s_t, a_t) = Q^B_t(s_t, a_t) + \alpha Q(s_t, a_t) \cdot (r_t + \gamma Q^A_t \cdot (s_{t+1}, \max_a Q^B_t(s_t + 1, a) - Q^{B_t}(s_t, a_t))$$

Recentemente le tecniche di Deep Q-Learning sono state accostate a quelle di Double Q-Learning e si è visto un miglioramento delle prestazioni rispetto al Q-learning classico. Questo accostamento è stato nominato Double DQN.[7]

Capitolo 3

Problema Specifico

3.1 Introduzione

Il nostro obiettivo è quello di allenare un agente a giocare a Tetris. Ci si pone come obiettivo dell'allenamento quello di vedere l'agente imparare a giocare non conoscendo l'ambiente e le regole che lo comandano.

3.2 Tetris

Tetris è un videogioco inventato da Aleksej Leonidovič Pažitnov e divenuto famoso a fine anni ottanta. Sviluppato per quasi tutte le piattaforme di gioco, ha ottenuto il secondo posto nella classifica dei 10 videogiochi più importanti al mondo. I pezzi del Tetris sono chiamati tetramini; ognuno di questi pezzi è composto da quattro blocchi (da cui deriva il nome tetramini). I tetramini cadono dall'alto uno alla volta e il giocatore può ruotarli verso destra, verso sinistra e spostarli nello spazio; il gioco si sviluppa in una griglia inizialmente vuota.

Il compito del giocatore è quello di completare una linea orizzontale di blocchi, senza interruzioni al suo interno. Oltre le linee, alcune versioni di tetris, offrono

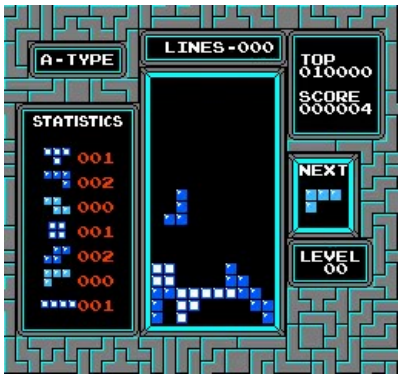


Figura 3.1: Schermata del gioco Tetris per Nintendo Entertainment System

dei punti per spingere verso il basso i tetramini. Alcune versioni offrono dei punti anche per il numero di tetramini posizionati attraverso la spinta verso il basso.

I pezzi vengono estratti casualmente dal gioco, e ogni gioco implementa degli algoritmi di estrazione propri, con lo scopo di rendere la partita più scorrevole. Quando il giocatore riempie tutta la griglia senza poter posizionare più tetramini la partita si interrompe.

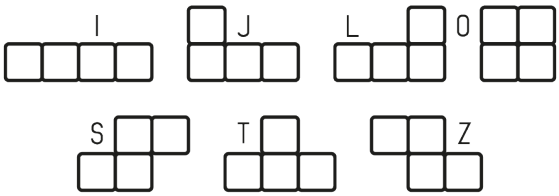


Figura 3.2: I Tetramini

3.2.1 Modalità di Gioco

Le varie versioni di tetris presentano diverse modalità:

- **A-Type** (Maratona) :

  Inizialmente la *board* è vuota. La velocità di caduta dei tetrimini aumenta

ogni 10 righe completate. Lo scopo di questa modalità è sopravvivere più tempo possibile cercando di massimizzare i punti ottenuti.

- **B-Type** :
  I pezzi cadono a velocità costante per tutta la partita ma avremo le righe della *board* parzialmente riempite da quadratini. La partita terminerà positivamente quando il giocatore avrà totalizzato 25 righe mentre avverrà una sconfitta quando il giocatore sarà impossibilitato a posizionare altri pezzi.

Entrambe le modalità solitamente hanno un livello di sfida selezionabile prima della partita, ma nella modalità maratona l'aumento della difficoltà e comunque legato al numero di linee totalizzate.

3.3 Rapporto Agente Tetris

Come spiegato nel capitolo 2.2 l'agente ha bisogno di uno stato in input per poter effettuare delle decisioni. Nel nostro problema, all' agente forniremo la griglia di gioco e le varie statistiche che il gioco mostra a schermo, contenute in un frame del gioco. L'interazione col gioco, invece, avverrà tramite il controller, che consentirà all'agente di effettuare tutte le mosse disponibili. Poichè il gioco è stato originariamente sviluppato per avere un output video analogico, con formato NTSC[28], si è deciso di utilizzare un emulatore per permettere di avere una riproduzione digitale della schermata di gioco. Un emulatore è un programma che permette di replicare funzioni di un determinato sistema su un secondo sistema diverso da quello di partenza garantendo, auspicabilmente con gli stessi risultati (ma non sempre questo è possibile), l'esecuzione di applicativi sviluppati per il sistema iniziale.[22] Generalmente, la digitalizzazione del Nintendo Entertainment System viene sviluppata in grado di fornire circa 30 frame al secondo per mantenere la fedeltà col formato NTSC.

Questo approccio facilità sia l'acquisizione delle informazioni sullo stato da fornire all'agente, che l'esecuzione delle azioni da parte dell'agente stesso.

3.4 Considerazioni

È dimostrato anche che in Tetris è possibile generare una sequenza di pezzi che permette la fine della partita giocata in una griglia di larghezza $2(2n + 1)$, con n intero. Questo implica che anche se si tentasse di allungare una partita a tempo infinito si arriverebbe inevitabilmente alla sconfitta.[5]

È stato dimostrato da Breukelaar[3] che Tetris è un problema NP-completo; NP è una classe di problemi le cui soluzioni si trovano in tempo polinomiale attraverso una macchina di Turing non deterministica. Più precisamente, Tetris è caratterizzabile come un insieme di sottoproblemi NP-completi: massimizzare il numero delle linee fatte, massimizzare il numero di tetris fatti, minimizzare l'altezza massima occupata da un tetramino e massimizzare il numero di tetramini posizionati prima della fine della partita. Il fatto che sia riconducibile ad un problema NP-complesso comporta l'impossibilità di trovare in tempo lineare una politica che riesca a trovare la migliore azione. Alcuni di questi sottoproblemi possono essere risolti utilizzando delle tecniche di approssimazione lasciando al reinforcement learning il compito di determinare delle politiche ottimali.

Data una sequenza di tetramini p è stata dimostrata l'estrema inapprosimabilità del sottoproblema del massimo numero di linee fatte e la massima sequenza di pezzi posizionati prima della fine della partita.[5]

Il numero di possibili stati in cui il gioco si può trovare è 7×2^{200}; questo fa sì che sia di fatto impossibile trovare una policy esatta per la soluzione.[19]
Data la complessità di questi giochi si è pensato che il Reinforcement Learning possa essere un'ottima soluzione per allenare delle Intelligenze Artificiali a giocarli.

Una delle assunzioni del Reinforcement Learning è che l'ambiente rispetti le caratteristiche del processo di Markov.[21] Tetris soddisfa questo requisito in quanto ogni informazione richiesta per decidere è rappresentata in ogni stato in cui si trova la partita.

Capitolo 4

Strumenti

4.1 Presentazione

Per realizzare l'esperimento si è scelto di utilizzare una serie di tool basati su Python 3.6, che è stato usato come linguaggio di programmazione. Le librerie utilizzate sono le seguenti:

- Gym Tetris[9] : Sviluppata da Christian Kauten e messo a disposizione nel gestore pacchetti di Python, è basata sulla libreria Open AI. L'enviroment è così strutturato: un emulatore inserito in un ambiente generico per agenti a cui sono state riscritte alcune funzioni per permettere la compatibilità con il gioco tetris per NES.

- Stable Baselines[8] : Libreria scritta partendo dal progetto OpenAI, Baselines offre la possibilità di creare agenti per il reinforcement learning e di interfacciarli con molti ambienti.

4.2 OpenAI Gym

Open AI è un'organizzazione di ricerca sull'intelligenza artificiale con lo scopo di sviluppare un'intelligenza artificiale amichevole (Friendly AI) in modo che l'umanità possa trarne beneficio.[4].

Queste "palestre per intelligenze artificiali" sono state anche realizzate per cercare di risolvere due problemi principali:

La necessità di ottenere migliori benchmark:

La difficoltà effettiva si aveva nel non avere sempre degli ambienti riproducibili all'infinito. OpenAI sta cercando di offrire un pacchetto di gym eterogeneo e che permetta la riproducibilità dei test così da avere termini di paragone per esperimenti nel reinforcement learning.

Assenza di standardizzazione degli ambienti usati nelle pubblicazioni:

Nelle pubblicazioni ci sono molte differenze su come vengono definiti i problemi, le funzioni di reward e lo spazio delle azioni. Questo può portare a delle difficoltà nella riproduzione, nell'ampliamento o nella comparazione tra più progetti.

4.2.1 Gym

Di seguito spiegheremo i metodi che permettono il funzionamento delle gym. Lo scheletro viene riportato nell'appendice al listato 7.1.

step(int: action **)**

Il metodo si aspetta un azione da parte dell'agente. Solitamente l'azione è di tipo int.

Return:

- observation (object) : Contiente un'istantanea dell'enviroment
- reward (float) : contiene il reward fornito dall'enviroment

- done (bool) : *True* se l'episodio è terminato, *False* in caso contrario
- info (dict) : Contiene delle informazioni inerenti all'enviroment solitamente utilizzate per debug ma può succedere che qualche gym passi dati utili al learning

reset(str: mode = 'human' **)**
Resetta lo stato dell'enviroment alla situazione iniziale
Return:

- observation (object): Un'istantanea della situazione iniziale

render(None **)**
Effettua il render dell'enviroment. Ci sono tre possibili opzioni di visualizzazione predefinite:

- human : Renderizza a schermo o sul terminale in base al tipo di enviroment. Come il nome lascia intuire è pensato per essere utilizzato in caso di necessità di visualizzazione.
- rgb_array: Restituisce un numpy.ndarray con shape (x , y , 3) che rappresenta dei valori RGB di un immagine con dimensione x , y. Può essere utilizzato per essere tramutato in video.
- ansi : Restituisce una stringa contenente una rappresentazione dell' enviroment. Viene reso noto che il testo potrebbe contenere caratteri di "presentazione" per rendere la visualizzazione più gradevole.

Return:

- observation (object): Un'istantanea della situazione iniziale

In seguito sono illustrate le differenze delle librerie con lo scheletro di base delle gym.

4.2.2 Gym Tetris

L'ambiente eredita le proprietà da un enviroment "generale", nes-py. A sua volta Nes-py eredita la struttura dalle gym fornite da Open AI. Nes-py è un enviroment al cui interno troviamo l'emulatore SimpleNes che permette l'esecuzione di giochi. Tramite alcune modifiche si può modellare un enviroment specifico per il gioco, grazie alle quali l'agente può interagire con esso. Gym-tetris permette di dichiarare, nel file actions.py, un set di azioni personalizzate per l'agente, che sono poi convertite da nes-py in comandi da mandare all'emulatore.
Si ricorda (o rende noto) al lettore, che i comandi di input inviati al NES attraverso il controller vengono gestiti direttamente dalla console andando a modificare una dato indirizzo di memoria.

Come specificato nel paragrafo 3.2.1, il gioco offre due modalità di gioco, A-type e B-type. La modalità A-type è quella utilizzata durante l'addestramento dell'agente e rispecchia la "modalità maratona", mentre il B-type è la "modalità rompicapo" in cui l'agente deve completare un certo numero di linee per vincere.

Nella modalità di gioco maratona l'enviroment offre tre tipi di reward da dare all'agente:

- Cambiamento nello score
- Cambiamento nel numero di linee completate
- Penalità per il raggiungimento di una certa soglia di altezza nella schermata di gioco.

Enviroment	Reward Punteggio	Reward Linee Fatte	Penalizzazione Altezza
TetrisA-v0	V	x	x
TetrisA-v1	x	V	x
TetrisA-v2	V	x	V
TetrisA-v3	x	x	V
TetrisB-v0	V	x	x
TetrisB-v1	x	V	x
TetrisB-v2	V	x	V
TetrisA-v3	x	V	V

Tabella 4.1: Le varie modalità con le rispettive configurazioni di reward per l'agente

4.2.3 Gym Tetris code

Andremo ora a spiegare le porzioni di codice di interesse, si farà richiamo ad esse e saranno disponibili nell'appendice.

tetris_env.py

Questo file contiene tutte le parti principali dell'ambiente. Andremo ad analizzarne i componenti principali. Reperibili nell'appendice al listato 7.2 .

La lista corrispondente all'insieme di tutte le possibili combinazioni pezzo - orientamento possibili. L'indice di ogni singola stringa corrisponde al valore numerico con cui il gioco rappresenta tale combinazione.

Current Piece

I dati relativi ai pezzi sono salvati nella ram del NES, si prende quindi il valore del pezzo corrente da essa, un intero, per poi usarlo come indice nella lista vista in precedenza. È presente il listato 7.3 nell'appendice.

Calcolo del Reward

L'enviroment restituisce il reward all'agente dopo ogni step. Tramite gli if controlla l'attuale configurazione dell'enviroment e applicarne i rispettivi metodi di calcolo

del punteggio. Nel penalizzare l'altezza, si controlla se l'altezza attuale è più alta dello step precedente ed, in caso lo sia, viene applicata al reward una penalità pari alla differenza delle due altezze. Rimando al listato 7.4 nell'appendice.

Get Info

L'ambiente restituisce anche le informazioni interenti allo stato attuale dell'esecuzione da restituire all'agente dopo ogni mossa eseguita. La funzione viene mostrata nel listato7.6 presente nell'appendice.

Sotto illustriamo brevemente in forma tabellare i valori restituiti.

Key	Tipo	Descrizione
current_piece	str	pezzo corrente
number_of_lines	int	numero di linee fatte
score	int	punteggio corrente
next_piece	str	pezzo successivo
statistics	dict	riporta le statistiche dei pezzi nella partita

Tabella 4.2: Informazioni passate all'agente

actions.py

In questo file troviamo le varie azioni disponibili, il valore testuale corrisponde al relativo tasto del controller. Così di seguito schematizzati:

- A
- B
- start
- select
- up
- down
- right
- left

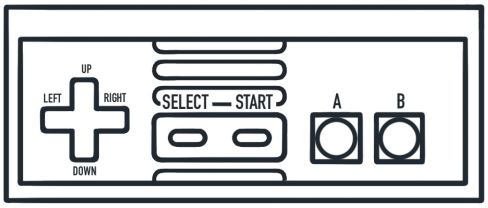


Figura 4.1: Schematizzazione del controller del NES

Nelle righe presenti nell'appendice al listato 7.7 è riportata la lista di mosse utilizzabili dall'agente in uno dei test effettuati. Le azioni sono state ridotte in modo da lasciare solo quelle indispensabili all'agente per giocare correttamente, così da poter lasciare spunti per un'eventuale aggiunta di mosse (o una combinazione di esse) e la rispettiva risposta dell'agente a esse.

4.3 Stable Baselines

Stable Baselines è un insieme di algoritmi per il Reinforcement Learning (RL) basato su OpenAI Baselines.[8] La libreria offre delle migliorie nelle implementazioni e anche un "RL Baselines Zoo", un insieme di agenti pre-addestrati messi a disposizione da utenti e sviluppatori. La libreria offre anche delle interfacce semplici per fare l'addestramento, la valutazione o il tuning degli iper-parametri degli agenti.
Il fork è nato a causa di alcune problematiche legate alla scarsa documentazione e alla macchinosità delle definizioni per l'addestramento degli agenti nella libreria OpenAI originale.

Implementazioni

La libreria offre l'implementazione di una buona varietà di algoritmi di Reinforcement Learning:

- Synchronous Advantage Actor Critic (**A2C**)
- Due versioni di Proximal Policy Optimization (**PPO**)
- Trus Region Policy Optimization (**TRPO**)
- Deep Q Network (**DQN**) e la sua versione Double Agent Deep Q Network
- Deep Deterministic Policy Gradient (**DDPG**)
- Actor Critic using Kronecker-Factored Trust Region (**ACKTR**)
- Efficient Actor-Critic with Experience Replay (**ACER**)

Come già spiegato nella prima parte di quest'elaborato si è scelto di usare **DQN** nelle sue varie versioni; andremo ora ad analizzare nel dettaglio alcune parti dell'implementazione.

4.3.1 Deep Q Network

La libreria offre di default il Double Q Learning e il Dueling attivi ma sono stati lasciati disattivati nella prima parte del test.

Policy

Le policy offerte sono le seguenti:

- **MlpPolicy**:
  Multi-Layer Perceptron; questa policy si basa su un modello di rete neurale artificiale con 2 livelli composti da 64 nodi.

- **LnMlpPolicy**:
  Multi-Layer Perceptron con Layer Normalisation la policy si basa sullo stesso modello della precedente ma tra i due layer principali c'è un layer di normalizzazione.

- **CnnPolicy**:
  Convolutional neural network. Questa policy implementa la critica dell'agente tramite una Rete Neurale Convoluzionale.

- **LnCnnPolicy**:
  Convolutional neural network con Layer Normalisation : Convolutional neural network. Questa policy implementa la critica dell'agente tramite una Rete Neurale Convoluzionale aggiungendo un livello di normlaizzazione tra i livelli.

Poiché il nostro enviroment offre la possibilità di ottenere il frame corrente, si è deciso di utilizzare la LnCnnPolicy inquanto una rete convoluzionale si presta bene all'analisi di immagini.

Metodi della classe

Di seguito riportiamo le funzioni di maggior interesse della classe.
Visto l'elevato numero dei parametri per ogni metodo nel codice, si riportano solo quelli di maggiore interesse nella forma:

Nome Parametro (*Tipo*: Valore Predefinito)

DQN Constructor

```
1 classstable_baselines.deepq.DQN( ** )
```

- **policy** (*DQNPolicy or str*: necessario) : Oggetto policy visto in precedenza
- **env** (*Gym environment or str*: necessario) : Come visto in precedenza Gym Tetris
- **gamma** (*float*: 0.99) : Fattore di sconto, influenza la golosità dell'agente nell'ottenere il reward
- **learning_rate** (*float*: 0.0005) : è il valore ϵ da passare all'ottimizzatore Adam [26]
- **buffer_size** (*int*: 50000) : Grandezza del buffer, utilizzato come memoria dell'agente, contenente i dati degli step
- **train_freq** (*int*: 1) : Questo parametro corrisponde al numero degli step che devono passare prima di aggiornare il modello
- **batch_size** (*int*: 32) : Numero di step che vengono salvati nel buffer
- **double_q** (*bool*: True) : Se True abilita Il Double-Q learning
- **learning_starts** (*int*: 1000) : Numero di step in cui il modello osserve prima che inizi ad imparare

- **target_network_update_freq** (*int*: 500) : Questo parametro corrisponde al numero degli step che devono passare prima di aggiornare la rete neurale
- **prioritized_replay** (*bool*: False) : Se True attiva la modalità di priorità del replay
- **verbose** (*int*: 0) : Valore che rappresenta l'output durante il learn. O nessun output, 1 informazioni sul training, 2 informazioni di debug di tensorflow
- **tensorboard_log** (*str*: None) : Questo parametro contiene la path della cartella dove verranno salvati i dati per Tensorboard. Se None/False disattiva il salvataggio dati per Tensorboard.
- **seed** (*int*: None) : Valore numerico per il generatore pseudo-random . Se None usa un seed random.
  Per un risultato deterministico si deve utilizzare *n_cpu_ts_sess* ad 1
- **n_cpu_ts_sess** (*int*: None) : Numero di TensorFlow Thread utilizzati. Se None, viene usato il numero di cpu della macchina

DQN Learn

```
1 DQN.learn( ** )
2
3 return (BaseRLModel)
```

- **total_timesteps** (*int*: Necessario) : Numero di steps per l'addestramento. Un numero maggiore migliorerà notevolmente l'apprendimento.
- **callback** (*(Union[callable, [callable], BaseCallback])*: None) : Funzione che viene richiamata ad ogni step a cui viene passato lo stato attuale dell'esecuzione. Alla funzione vengono implicitamente passati due parametri:

_locals : Contiene le variabili dell'algoritmo

_globals : Contiene delle informazioni generali sul modello

Si possono usare i dati ottenuti per valutare l'andamento generale del learn e se fosse necessario anche interromperlo tramite il return. Ritornare il valore True proseguirà l'esecuzione mentre con False il learn si interrompe.

- **log_interval** (*int*: 100) : Numero di step prima di iniziare il log
- **tb_log_name** (*str*: 'DQN') : Il nome della run per il log di Tensorboard, seguito dal numero dell'episodio.

Gli ultimi due parametri anche se impostati vengono ignorati se nella dichiarazione del model non viene attivato il log di Tensorboard.
Il return aggiorna il modello con quello appena trainato.

DQN save

```
1 DQN.save( ** )
```

- **save_path** (*str or file-like*: Necessario) : Nome del modello da salvare, si può decidere di aggiungere anche una path
- **cloudpickle** (*bool*: False) : Salva il modello utilizzando la libreria *cloudpickle* invece che salvare l'archivio.

DQN Load

```
1 DQN.load( ** )
```

- **load_path** (*str or file-like*: Necessario) : Nome del modello salvato, si può decidere di aggiungere anche una path

- **env** (*Gym Enviroment*: None) : Passa l'enviroment su cui lavorare al modello. Se *None* si limiterà a fare un predict usando il modello e i dati in input su *predict()*

DQN predict

```
1 DQN.predict( ** )
2
3 return (np.ndarray, np.ndarray)
```

- **observation** (*np.ndarray*: Necessario) : Il frame attuale dell'enviroment
- **state** (*np.ndarray*: None) : Lo stato attuale dei dati dell'enviroment

Il metodo ritorna l'azione da eseguire e lo stato successivo dell'enviroment.

DQN step

```
1 DQN.step( ** )
2
3 return (np.ndarray int, np.ndarray float, np.ndarray float)
```

- **observation** (*np.ndarray*: Necessario) : Il frame attuale dell'enviroment
- **state** (*np.ndarray*: None) : Lo stato attuale dei dati dell'enviroment

il metodo ritorna rispettivamente l'azione, il q_values legato all'azione e lo stato attuale.

Capitolo 5

Design Degli Esperimenti

In questo capitolo verrà spiegato nel dettaglio il rapporto tra l'agente e l'ambiente, introdurremo la struttura del processo di sperimentazione e spiegheremo il perché di tali scelte.

5.1 Rapporto Agente-Ambiente

Seguendo lo schema della gym la funzione di *learn()* permette l'esplorazione dell'ambiente all'agente. Il modello riceve un oggetto che rappresenta la schermata di gioco inerente allo stato successivo; questa schermata viene convertita da un immagine RGB in un array multidimensionale e multilivello. Più precisamente, l'immagine viene divisa in tre livelli, uno per ogni canale cromatico, e di larghezza e altezza pari alla schermata del gioco. I dati satellite inerenti alla partita, invece, sono passati all'agente come tupla contenente delle stringhe testuali. Una volta che l'agente ha ottenuto l'informazione dello stato corrente effettua una previsione tramite *prediction()* e restituisce un valore numerico di tipo intero, rappresentante l'indice della mossa da utlizzare, all'ambiente che poi converte il valore in mossa e la esegue.

5.2 Parte Progettuale

Principalmente il progetto si divide in due parti. La parte di learn e la parte di test. Di seguito analizzeremo solo le parti più importanti pertanto le dichiarazioni dell'enviroment e i vari import verranno omessi. Tutto il codice del progetto verrà comunque messo nell'apposita sezione dell'appendice, sia in versione ipython sia in versione python.

5.2.1 Addestramento

Nella parte di addestramento si sono sviluppate alcune funzioni per il monitoraggio dell'esecuzione utilizzando la funzione di callback.
Questa fase, segue un flusso di esecuzione che possiamo così riassumere:

- Dichiarazione del modello e di eventuali parametri di tuning
- Addestramento (learning), in cui si consente all'agente di apprendere per un dato numero di steps
- Salvataggio del modello

```
1 model = DQN(CnnPolicy, env,verbose=1)
2 model.learn(total_timesteps=20000, callback=callback)
3 model.save("Model_Name")
```

La possibilità di avere salvataggio e caricamento è utile per poter effettuare il learn in più fasi. Banalmente i tempi sono dettati dall'interazione con l'enviroment e dal numero di steps che si vuole che l'agente faccia. Segmentare la fase di learning ci permette di analizzare l'andamento più frequentemente senza dover aspettare lunghi tempi di addestramento.

Callback

Analizziamo ora la funzione di callback utilizzata, in riferimento al listato 7.5 nell'appendice.

La variabile *_locals* è un array associativo al cui interno sono contenute tutte le informazioni relative all'enviroment. Col primo if ogni 5000 step si stampa a video un breve riassunto delle informazioni principali, mentre con il secondo si salva un riassunto generale del learn in un file di testo tramite la funzione *write_log()*. L'utente può personalizzare a proprio piacimento la funzione; si ricorda che è possibile eseguire valutazioni *"on the go"* e interromppere l'addestramento in qualsiasi momento. In questo caso si è omesso di valutare l'andamento dell'addestramento perché si è preferito fare periodiche analisi dei modelli tramite delle simulazioni di partite all'agente.

5.2.2 Test

Come per la parte di learn si propone un flusso logico della sessione di test.

- Caricamento del modello
- Esecuzione degli episodi
- Valutazione dell'andamento medio tramite i reward

```
1 model = DQN.load("Model_Name", env=env)
2 mean_reward = evaluate(model=model, env=env,
    episode=num_episodes)
```

L'esecuzione della funzione *evaluate()* segue lo scheletro delle gym proposto nella presentazione di esse. È mostrato approfonditamente nell'appendice al listato 7.9. Le modifiche che sono state effettuate sono state la creazione di una funzione *evaluate()* a cui si passa il numero degli episodi il modello e se si deve o non deve effettuare il render a video degli esperimenti. All'interno dello scheletro della gym, l'agente restituisce l'azione da effettuare tramite la funzione *model.predict(obs)* a cui si passa lo stato attuale dell'ambiente. È stato anche utilizzato un metodo per calcolare il valore medio dei reward di ogni episodio.

5.3 Esperimenti

Per realizzare gli esperimenti si è deciso di effettuare tre prove distinte, con ognuna di esse mirata a vedere il comportamento dell'agente tramite dei cambi di enviroment o di agente. I modelli dei tre esperimenti sono stati allenati da zero per riuscire a trovare una configurazione che facilitasse l'apprendimento.

Nel primo esperimento si sono lasciati i valori dei parametri offerti di default da Stable-Baselines[8].

Il secondo esperimento è stato svolto andando a modificare i parametri di tuning. I parametri sono reperibili al listato 7.11 nell'appendice. Seguendo quanto riportato nel capitolo 2 si è pensato di utilizzare il Double Q-Learning per cercare di migliorare le prestazioni, anche a discapito del tempo di allenamento. In base a quanto offerto dalla libreria si è pensato di dare priorità al replay dell'esperienza perché questo tende ad aumentare le capacità di apprendimento. Come dimostrato in "*Prioritized Experience Replay*" [18] questa esperienza acquisita aggiuntiva tende a rinforzare il Q-learning fissando in maniera ancora più decisa i punti forti delle scelte effettuate.

Nel terzo esperimento, in relazione ai test effettuati nel secondo è stato deciso di modificare la lista di azioni utilizzabili dall'agente 7.8 mantenendo però i parametri di tuning utilizzati nel secondo esperimento.7.11.

Nel prelevare i dati dalle prove dell'agente si è deciso di fare due partite consecutive per ogni fetta di step dell'allenamento del modello. Per prelevare questi dati si è utilizzato il metodo presente nell'appendice al listato 7.12. Questo metododo è prsonalizzato per scrivere un file di log che contenesse i dati di ogni prova. Per ogni prova carica i modelli delle varie fasi di allenamento e gli fa giocare due partite.

Per avere anche un riscontro visivo delle partite è stato creato uno script che permettesse di renderizzare un video delle partite. Il metodo si può vedere nell'appendice al listato 7.14.

Tutto l'esperimento, una breve guida introduttiva e tutte le risorse utilizzate per questa tesi sono presenti nella mia personale repository github. [16]

Capitolo 6

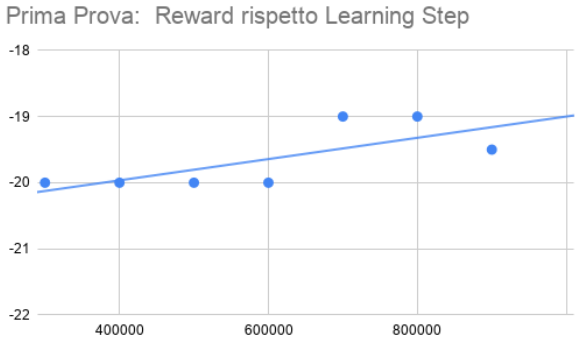
Esperimenti

6.1 Primo Esperimento

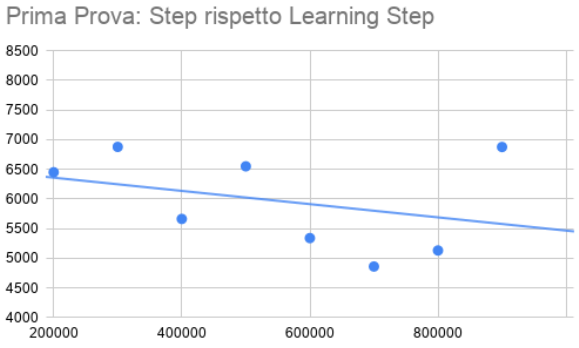
I tempi di addestramento del modello definito per questo esperimento sono stati veloci, ma hanno portato a scarsi risultati rispetto alla risoluzione di tutti i problemi espressi nel capitolo 3. Nonostante la scelta di un elevato numero di training step, pari ad un milione, non si è comunque riusciti ad ottenere un risultato apprezzabile. L'agente addestrato con questo approccio adotta una strategia che lo porta ad impilare i tetramini verso uno dei due lati della griglia cercando di muovere i pezzi ignorando completamente la possibilità di fare le linee. Un aumento del tempo di sopravvivenza nella partita era l'obiettivo minimo che ci si era posti, ma la prova è risultata inconcludente.

Learn Step	Reward Medio	Step Medi
200000	-20	6890
300000	-20	6453
400000	-20	6881
500000	-20	5666
600000	-20	6553
700000	-20	5343
800000	-19	4863
900000	-19	5134
1000000	-19,5	6879

Tabella 6.1: Dati Primo Esperimento



(a) Grafico : Reward Medio / Learning Steps



(b) Grafico : Step Medi / Learning Steps

Figura 6.1: Dati inerenti al primo test

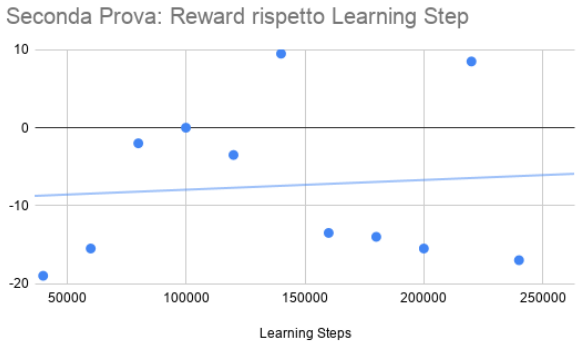
6.2 Secondo Esperimento

In questo esperimento, i tempi di addestramento rispetto al numero degli step sono aumentati notevolmente. L'agente ha aumentato, però, notevolmente il punteggio ottenuti durante la partita e le sue performance di gioco rispetto al tempo di sopravvivenza, ma si è fossilizzato sullo spingere verso il basso i tetramini per ottenere punti, ignorando completamente la possibilità di fare punti attraverso le linee.

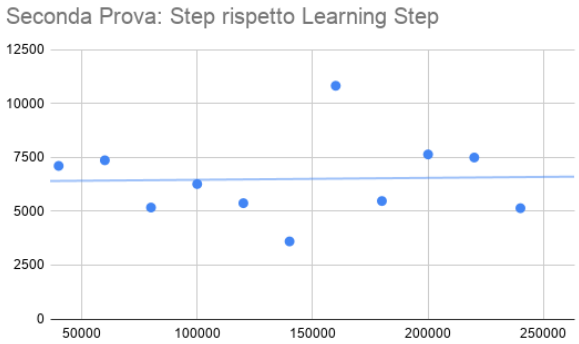
Learning Steps	Score Medio	Reward Medio	Step Medi
20000	0	-20	5951
40000	1	-19	3300
60000	1	-19	7110
80000	4,5	-15,5	7371
100000	18	-2	5178
120000	20	0	6267
140000	16,5	-3,5	5380
160000	29,5	9,5	3604
180000	6,5	-13,5	10829
200000	6	-14	5480
220000	4,5	-15,5	7643
240000	28,5	8,5	7498
260000	3	-17	5147

Tabella 6.2: Dati Secondo Esperimento

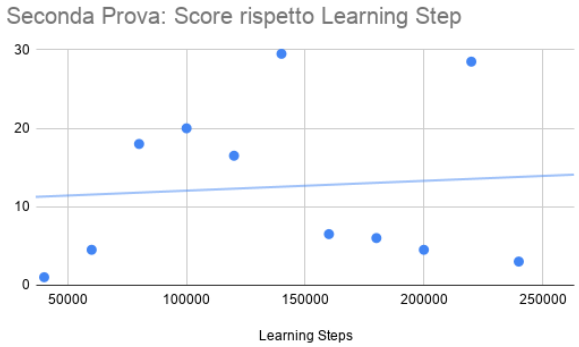
In conclusione questa strategia "vincente" ha fatto sì che l'agente si concentrasse sullo spingere un numero più alto di tetramini verso il basso cercando, quasi, di spostare dove ci fosse più spazio libero.



(a) Grafico : Reward Medio / Learning Steps



(b) Grafico : Step Medi / Learning Steps



(c) Grafico : Punteggio Medio / Learning Step

Figura 6.2: Dati inerenti al secondo test

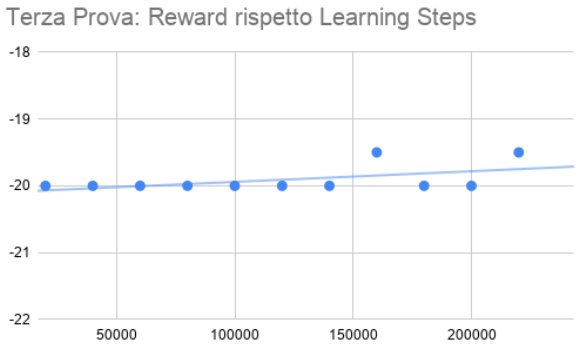
6.3 Terzo Esperimento

In questa prova di apprendimento all'agente è stata tolta la possibilità di utilizzare il pulsante "down". Questa scelta è stata fatta per cercare di prendere i buoni risultati ottenuti con la modifica dei parametri di tuning evitando, però, che si creasse la strategia per ottenere il punteggio dell'esperimento precedente.

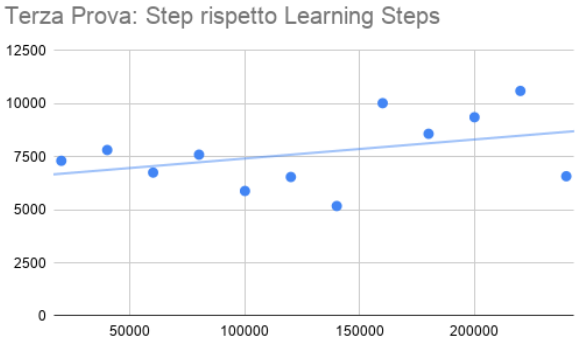
Learning Steps	Reward Medio	Step Medi
20000	-20	7307
40000	-19	7816
60000	-20	6752
80000	-20	7596
100000	-20	5886
120000	-20	6544
140000	-20	5177
160000	-20	10021
180000	-19	8579
200000	-20	9358
220000	-20	10597
240000	-19	6580

Tabella 6.3: Dati Terzo Esperimento

Come previsto, l'agente ha aumentato i tempi relativi alla sopravvivenza ma non è riuscito ad apprendere il concetto di linea e, perciò, di punteggio. Anche se in fase di allenamento un numero notevole di linee è stato completato, questo non è stato sufficente per permettere la propagazione all'indietro dell'esperienza della sequenza di mosse che ha portato a farlo.



(a) Grafico : Reward Medio / Learning Step



(b) Grafico : Step Medi / Learning Step

Figura 6.3: Dati inerenti al terzo test

6.4 Anomalie di learn e di test

In alcune fasi di allenamento degli esperimenti si è verificato che l'agente eseguisse delle linee o aumentasse i tempi di sopravvivenza in maniera notevole, ma questo non ha avuto riscontri nelle fasi di test. Queste anomalie si sono riscontrate anche nelle fasi di test, ma ripetendo le prove con la stessa conformazione si è verificato come questi fossero dei piccoli casi isolati. Questo potrebbe verificarsi come contrapposizione a quanto descritto nel capitolo 3 in quanto potrebbero susseguirsi delle sequenze di pezzi che favoriscono la sopravvivenza meglio di altre.

Capitolo 7

Conclusioni

Nelle prove si sono visti dei miglioramenti grazie al tuning dei parametri. Nel secondo esperimento il miglioramento della strategia di gioco finalizzata all'aumento della sopravvivenza è stato molto più immediato rispetto alla prima prova. Questo aumento nel tempo di sopravvivenza è coerente con i reward forniti dall'ambiente. Essi hanno favorito l'aumento graduale del tempo di gioco nelle partite in concomitanza dell'aumento del tempo di allenamento. L'aumento del tempo di sopravvivenza, nella seconda prova, è stato seguito anche da un aumento del punteggio di gioco. L'aumento di punti fatti, però, non è legato al numero di linee completate ma allo spingere verso il basso i tetramini portando l'agente a cercare sia di spingerli che a non impilarli troppo. Nella terza prova è stata tolta la possibilità di spingere i pezzi verso il basso ma come riscontro si è ottenuto solo l'aumento del tempo di sopravvivenza. Questo ci ha fatto capire che l'agente è in grado di adattarsi all'ambiente e di trovare strategie risolutive anche non banali. Possiamo anche dire che l'agente nella seconda e nella terza prova ha comunque cercato di risolvere il problema del massimizzare il numero di tetramini posizionati prima della fine della partita. Non si sono ottenuti risultati necessari a poter considerare la strategia vincente ma sono comunque degni di nota.

Tuttavia l'esperimento ha dimostrato quanto detto in *The Game of Tetris in*

Machine Learning[19]. Tetris a causa della sua natura non è un gioco che si presta all'apprendimento per rinforzo. Necessita di strategie decisionali e di sopravvivenza che prevedano tutti i possibili posizionamenti del tetramino cercando di massimizzare i punti rispetto alla situazione attuale della partita. La propagazione all'indietro dei reward positivi non è abbastanza soddisfacente da poter creare delle esperienze di buon andamento, in quanto anche se si verificano dei picchi di punteggio dati dalla risoluzione di linee, sono deboli rispetto alla quantità di punteggio che si ottiene mandando i pezzi verso il basso.

Come suggerito in *The Game of Tetris in Machine Learning*[19] il Reinforcement è un metodo risolutivo inferiore rispetto agli algoritmi genetici, in quanto quest'ultimi selezionano i candidati ottimali al livello della popolazione, mentre il Reinforcement Learning opera tale selezione a livello individuale. Nel Reinforcement Learning, ogni azione compiuta è giudicata rispetto, e indirizzata verso l'azione ottimale nello stato. Di contro, gli algoritmi genetici premiano loci genetici completi a prescindere dal comportamento dei geni individuali nell'episodio precedente. Si pone anche l'attenzione sul fatto che sia stato dimostrato che tramite la semplificazione dell'ambiente si possa notevolmente aumentare l'apprendimento, ma trasferendo (tramite tecniche di Transfer Learning) quanto appreso non si hanno notevoli riscontri di miglioramento.[5]

7.1 Possibili soluzioni e lavoro futuro

Come spiegato nel 3 Tetris è diviso in sottoproblemi NP-completi, questo lo rende ancora un problema aperto per la comunità scientifica. In futuro si potrebbe pensare di focalizzare gli sforzi nel risolvere singoli sottoproblemi. Precisamente nel minimizzare l'altezza massima occupata da un tetramino e massimizzare il numero delle linee eseguite.

Un'altra proposta di soluzione al problema è quella di modificare l'ambiente in modo che restituisca dei punteggi negativi per la creazione di pozzi, buchi nel posizionamento dei tetrimini e che aumenti il punteggio ottenuto totalizzando delle linee. Potremmo definire questa modifica come una possibile simulazione di euristica, però all'interno dell'ambiente.
Si potrebbe, infine, anche applicare lo stesso approccio visto in Aplha Go.[24] Offrendo all'agente partite effettuate da altre persone sottoforma di associazione stato-mossa-punteggio-stato successivo. Per poi lasciare all'agente il miglioramento della tecnica di gioco tramite apprendimento per rinforzo.

Appendice Dei Codici

Listing 7.1: Struttura della gym

```
1  import gym
2  env = gym.make('CartPole-v0')
3  for i_episode in range(20):
4      observation = env.reset()
5      for t in range(100):
6          env.render()
7          print(observation)
8          action = env.action_space.sample()
9          observation, reward, done, info = env.step(action)
10         if done:
11             print("Episode␣finished␣after␣{}␣
                   timesteps".format(t+1))
12             break
13 env.close()
```

Listing 7.2: Array dei pezzi e dei possibili orientamenti

```
_PIECE_ORIENTATION_TABLE =
[
    'Tu','Tr','Td','Tl',         #T
    'Jl','Ju','Jr','Jd',         #J
    'Zh','Zv',                   #Z
    'O',                         #O
    'Sh','Sv',                   #S
    'Lr','Ld','Ll','Lu',         #L
    'Iv','Ih',                   #I
]
```

Listing 7.3: Funzione che prende dalla ram del gioco il pezzo corrente

```
    def _current_piece(self):
        """Return the current piece."""
        try:
            return _PIECE_ORIENTATION_TABLE[self.ram[0x0042]]
        except IndexError:
            return None
```

Listing 7.4: Funzione che restituisce il reward all'agente

```
1  def _get_reward(self):
2          """Return the reward after a step occurs."""
3          reward = 0
4          # reward the change in score
5          if self._reward_score:
6              reward += self._score - self._current_score
7          # reward a line being cleared
8          if self._reward_lines:
9              reward += self._number_of_lines -
                   self._current_lines
10         # penalize a change in height
11         if self._penalize_height:
12             penalty = self._board_height -
                   self._current_height
13             # only apply the penalty for an increase in
                   height (not a decrease)
14             if penalty > 0:
15                 reward -= penalty
16         # update the locals
17         self._current_score = self._score
18         self._current_lines = self._number_of_lines
19         self._current_height = self._board_height
20         return reward
```

Listing 7.5: Funzione di callback

```
1  def callback(_locals, _globals):
2      if _locals['_']%5000 == 0 and _locals['_'] != 0:
3          string = "Steps: " + str(_locals['_']) + "\n"
4          string = string + "Num_episodes: " + str(
               _locals['num_episodes']) + "\n"
5          string = string + "episode rewards: " +  str(
               _locals['episode_rewards']) + "\n"
6          string = string + "mean 100ep reward: " + str(
               _locals['mean_100ep_reward']) + "\n"
7          string = string + "Number of Lines: " +
               str(_locals['info']['number_of_lines'])
8          print(string)
9      if _locals['_'] == (last_step - 1):
10         string = "Steps: " + str(_locals['_']) + "\n"
11         string = string + "Num_episodes: " + str(
               _locals['num_episodes']) + "\n"
12         string = string + "episode rewards: " +  str(
               _locals['episode_rewards']) + "\n"
13         string = string + "mean 100ep reward: " + str(
               _locals['mean_100ep_reward']) + "\n"
14         string = string + "Number of Lines: " +
               str(_locals['info']['number_of_lines'])
15         print("Sto scrivendo il file di log")
16         write_log(string)
17     return True
```

Listing 7.6: Funzione che restituisce le informazioni sullo stato all'agente

```
def _get_info(self):
        """Return the info after a step occurs."""
        return dict(
            current_piece=self._current_piece,
            number_of_lines=self._number_of_lines,
            score=self._score,
            next_piece=self._next_piece,
            statistics=self._statistics,
            board_height=self._board_height,
        )
```

Configurazioni presenti nel file actions.py:

Listing 7.7: Lista dei tasti di base da far usare all'agente

```
TRAIN_MOVEMENT = [
    ['down'],
    ['A'],
    ['right'],
    ['left]
]
```

Listing 7.8: Azioni Terzo Episodio

```
    TRAIN_NODOWN = [
        ['A'],
        ['B'],
        ['right'],
        ['left'],
    ]
```

Listing 7.9: Funzione di Valutazione

```
def evaluate(model, env, episode, render = False):
    episode_rewards = [0.0]
    obs = env.reset()
    step = 0
    episode = 0
    while i <= episode:
        step=step+1
        action, _states = model.predict(obs)
        obs, rewards, dones, info = env.step(action)
        if render:
            env.render()
        episode_rewards[-1] += rewards[0]
        if dones[0]:
            episode=episode+1
            obs = env.reset()
            episode_rewards.append(rewards)

    mean_100ep_reward = np.mean(episode_rewards[-100:])

    return mean_100ep_reward
```

Dichiarazioni dei modelli per i tre esperimenti coi rispettivi parametri di tuning:

Listing 7.10: Parametri Di Tuning Primo Esperimento

```
1 #model = DQN(CnnPolicy, env, verbose=1)
```

Listing 7.11: Parametri Di Tuning Secondo Esperimento

```
1         model =
             DQN(CnnPolicy,env,verbose=1,learning_starts=1000,
2         double_q= True,target_network_update_freq=500,
3         prioritized_replay=True,n_cpu_tf_sess =1)
```

Listing 7.12: Data Plotter

```
1  def plotter(model, env,num, episode ):
2      obs = env.reset()
3      step = 0
4      episode_rewards = 0
5      string = ""
6      i = 1
7      while i <= episode:
8          step=step+1
9          action, _states = model.predict(obs)
10         obs, rewards, dones, info = env.step(action)
11         episode_rewards += rewards
12         string = ""
13         if dones[0]:
14             string = string + str(num) + "\n"
15             string = string + "ep␣,␣n_linee␣␣,␣␣␣score␣␣␣,␣
                   Env␣rewards␣,␣step␣fatti␣\n"
16             string = string + str(i) +  "␣,␣" +
                   str(info[0]['number_of_lines']) +  "␣,␣" +
17                 str(info[0]['score']) +  "␣,␣" +
                       str(episode_rewards[0]) +  "␣,␣" +
                       str(step) + "\n"
18             i=i+1
19             episode_rewards = 0
20             obs = env.reset()
21             write_log(string)
22             string = ""
23
```

```
24 for num in [200,300,400,500,600,700,800,900,1000]:
25     model = DQN.load("TetrisA-v2_DQN_" + str(num) + "k",
           env=env)
26     print("test␣num␣:␣" + str(num))
27     plotter(model=model, env=env,num=num, episode=2)
28     del model
```

Listing 7.13: Funzione che scrive su file le stringhe passate

```
1 def write_log(stringa):
2     with open("log.txt", "a") as file_object:
3         file_object.write(stringa)
4         file_object.close()
```

Listing 7.14: Video Recorder

```
def record(model, env, num_episodes=1):
  env = VecVideoRecorder(env,"./vid",
     record_video_trigger=lambda x: x == 0,
     video_length=12000, name_prefix="tetris_ai_video")
  obs = env.reset()
  i=0
  steps = 0
  while (i <= num_episodes):
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    steps+=1
    if dones[0]:
        obs = env.reset()
        print("=== EPISODE {} ===".format(i+1))
        print("Num_lines: " +
           str(info[0]['number_of_lines']))
        print("score: " + str(info[0]['score']))
        print("Number of episodes: ", i)
        print("=== END ===")
        i+=1
  return "Done"
```

Bibliografia

[1] «A Formal Measure of Machine Intelligence». In: *Proc. 15th Annual Machine Learning Conference of Belgium and The Netherlands (Benelearn 2006)* (), pp. 73–80.

[2] A. G. Barto, R. S. Sutton e C. W. Anderson. «Neuronlike adaptive elements that can solve difficult learning control problems». In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 834–846.

[3] Ron Breukelaar. «Tetris is Hard, Even to Approximate». In: (2004). URL: http://people.csail.mit.edu/dln/papers/tetris/tetris.pdf.

[4] Greg Brockman et al. *OpenAI Gym*. 2016. arXiv: 1606.01540 [cs.LG].

[5] Donald Carr. «Adapting Reinforcement Learning to Tetris». In: (dic. 2005).

[6] Arthur Guez Hado van Hasselt e David Silver. «Deep Reinforcement Learning with Double Q-Learning». In: ().

[7] Hado V. Hasselt. «Double Q-learning». In: *Advances in Neural Information Processing Systems 23*. A cura di J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

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

[9] Christian Kauten. *Tetris (NES) for OpenAI Gym*. https://github.com/Kautenja/gym-tetris. 2019.

[10] «Mastering the game of Go without human knowledge». In: *Nature* 550.7676 (2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[11] J.M. Mendel e R.W. McLaren. «8 Reinforcement-Learning Control and Pattern Recognition Systems». In: *Adaptive, Learning and Pattern Recognition Systems*. A cura di J.M. Mendel e K.S. Fu. Vol. 66. Mathematics in Science and Engineering. Elsevier, 1970, pp. 287–318. DOI: https://doi.org/10.1016/S0076-5392(08)60497-X. URL: http://www.sciencedirect.com/science/article/pii/S007653920860497X.

[12] Jerry M Mendel. «A survey of learning control systems(On-line-learning and off-line-learning self- organizing control systems)». In: *ISA TRANSACTIONS* 5 (1966), pp. 297–303.

[13] Donald Michie. «Experiments on the Mechanization of Game-Learning Part I. Characterization of the Model and its parameters». In: *The Computer Journal* 6.3 (nov. 1963), pp. 232–236. ISSN: 0010-4620. DOI: 10.1093/comjnl/6.3.232. eprint: https://academic.oup.com/comjnl/article-pdf/6/3/232/1030939/6-3-232.pdf. URL: https://doi.org/10.1093/comjnl/6.3.232.

[14] M. Minsky. «Steps toward Artificial Intelligence». In: *Proceedings of the IRE* 49.1 (1961), pp. 8–30.

[15] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[16] Mattia Polticchia. *Thesis Work*. https://github.com/TheTrash/Thesis-Work. 2020.

[17] Peter Russell Stuart; Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[18] Tom Schaul et al. *Prioritized Experience Replay*. 2015. arXiv: 1511.05952 [cs.LG].

[19] Özgür Şimşek Simón Algorta. «The Game of Tetris in Machine Learning». In: (). URL: https://arxiv.org/abs/1905.01652.

[20] Alex Smola e S.V.N. Vishwanathan. *Introduction to Machine Learning*. 2008.

[21] Richard S. Sutton e Andrew G. Barto. *Reinforcement Learning: An Introduction*. URL: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[22] Treccani. *emulatóre - Vocabolario on line*. URL: http://www.treccani.it/vocabolario/emulatore/.

[23] M. Waltz e K. Fu. «A heuristic approach to reinforcement learning control systems». In: *IEEE Transactions on Automatic Control* 10.4 (1965), pp. 390–398.

[24] Wikipedia. *AlphaGo*. URL: https://it.wikipedia.org/wiki/AlphaGo.

[25] Wikipedia. *Descrizione OpenAI*. URL: https://it.wikipedia.org/wiki/OpenAI.

[26] Wikipedia. *Discesa Stocastica del Gradiente*. URL: https://it.wikipedia.org/wiki/Discesa_stocastica_del_gradiente#Adam.

[27] Wikipedia. *Reinforcement learning*. URL: https://en.wikipedia.org/wiki/Reinforcement_learning.

[28] wikipedia. *NTSC*. URL: https://en.wikipedia.org/wiki/NTSC.