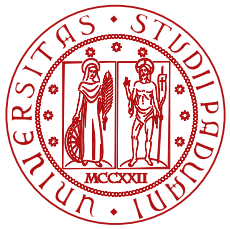



Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

*Dipartimento di Ingegneria dell'Informazione*

*Corso di Laurea in Ingegneria Informatica*


# Reinforcement Learning: studio e sviluppo di un IA che impari a giocare a Snake


Relatore:

**Prof. Luca Schenato**

Laureando:

**Ayoub Outmani**



Anno Accademico 2022/2023

Data di laurea 13/03/2023

# Abstract

Il reinforcement learning è un tipo di tecnica di machine learning che permette ad un agente di imparare a svolgere delle azioni utilizzando l'ambiente circostante e imparando, con vari tentativi, dalle proprie esperienze passate e dai propri errori. Uno degli esempi più eclatanti di applicazioni del reinforcement learning è AlphaGo un software che gioca a Go, sviluppato da Google DeepMind, che nel 2016 è riuscito a battere Lee Sedol, uno dei più grandi giocatori di Go. In questa tesi, dopo aver parlato della storia dell'intelligenza artificiale e del reinforcement learning e dopo aver trattato della teoria dietro al machine learning, ho proposto un software che impara autonomamente a giocare a Snake.

# Elenco delle figure

# Indice

# Capitolo 1

# Introduzione

## 1.1 La nascita dell'intelligenza artificiale

Sin dall'inizio del 20° secolo le persone hanno cominciato a scontrarsi con l'idea di macchina con un intelligenza paragonabile a quella umana, questo grazie ai romanzi, inizialmente, e ai film fantascientifici successivamente. Negli anni '50 diversi scienziati, matematici e filosofi erano già familiari con il concetto di intelligenza artificiale, uno tra questi fu Alan Turing, matematico e filosofo britannico che si interrogò sulla possibilità di applicare la matematica all'IA. Turing si chiese se in qualche modo le macchine, come gli umani, potessero utilizzare le informazioni a loro disposizione e il ragionamento per risolvere problemi e prendere decisioni. Nel suo articolo *Computing machinery and intelligence*, pubblicato nel 1950 sulla rivista *Mind* trattò come costruire macchine intelligenti e testare la loro intelligenza.
Vi erano però due criticità. La prima era che i calcolatori prima del 1949 non potevano memorizzare i comandi ma solo eseguirli, in quanto privi di una memoria. La seconda era che i computer all'epoca erano ancora estremamente costosi. C'era quindi bisogno di una Proof of Concept e anche di sostegno da persone facoltose per dimostrare l'utilità nel finanziare la ricerca sull'intelligenza artificiale.
La prima volta in cui la locuzione *Artificial Intelligence* venne usata ufficialmente fu nell'estate del 1956 alla *Dartmouth Summer Research Project on Artificial Intelligence*, una conferenza dove venne anche presentato *Logic Theorist*, un programma che riusciva a dimostrare 38 teoremi su 52 presi dal *Principia Mathematica* di Whitehead e Russell. La conferenza non ebbe risvolti eclatanti ma permise a diverse figure di spicco nella materia di conoscersi.

## 1.2 Storia dell'apprendimento per rinforzo

Il reinforcement learning, o apprendimento per rinforzo, ha le sue radici nel campo della psicologia. Infatti negli anni '30, lo psicologo statunitense Burrhus Frederic Skinner dimostrò che si può insegnare agli animali ad eseguire azioni complesse attraverso semplici meccanismi di rinforzo come ad esempio ricevere un premio, cibo in questo caso, dopo aver svolto l'azione richiesta. A seguito di svariati esperimenti su topi e piccioni, Skinner scoprì che si poteva plasmare il comportamento degli animali mediante il rinforzo, negativo o positivo. L'idea di base di quest'ultimo è che un animale o un generico agente possa imparare ad ottimizzare il suo comportamento utilizzando le proprie esperienze passate.

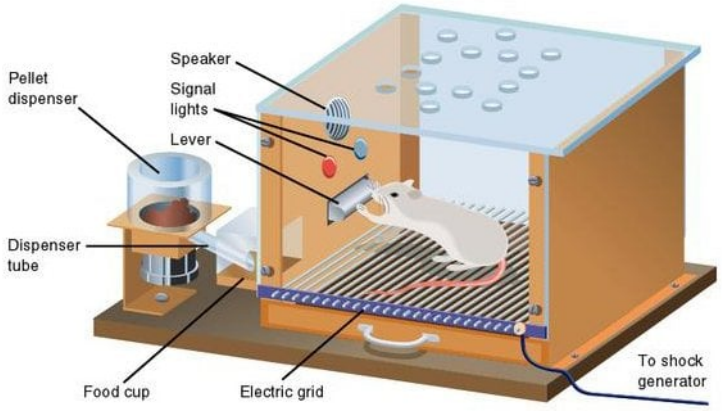



Figura 1.1: Gabbia di Skinner

Un'idea che era venuta ai ricercatori era quella di provare ad implementare un apprendimento per tentativi ed errori (trial and error) nei computer. Turing nel 1948 parlò in un articolo di un "pleasure-pain system": quando si giunge ad una configurazione per la quale l'azione non è determinata, viene effettuata una scelta casuale per il dato mancante, e viene successivamente applicata. Se viene ricevuto un responso negativo, tutte le voci vengono cancellate, in caso di responso poitivo vengono invece rese permanenti. Questa tecnica però si tramuto rapidamente negli anni in una ricerca di una generalizzazione e di un riconoscimento di uno schema, passando però da reinforcement a supervised learning e creando confusione sulla differenza tra le due tecniche di apprendimento.
Negli anni '50 e '60 i ricercatori cominciarono a sviluppare metodi di reinforcement learning da applicare all'IA. Richard Bellman fu il pioniere della programmazione dinamica, un insieme di tecniche volte a risolvere problemi di controllo ottimo. Per controllo ottimo si intende un insieme di equazioni, utilizzate per minimizzare o massimizzare diverse misure di un sistema dinamico nel tempo. A seguito delle sue ricerche, Bellman definì un insieme di equazioni funzionali, che prese il suo nome. La programmazione dinamica e le tecniche di apprendimento vennero sempre studiate separatamente in quanto applicate a materie diverse. Solo a partire dagli anni '80 questi due filoni si unirono per formare quello che è il moderno settore del reinforcement learning.

# Capitolo 2

# Apprendimento Automatico

## 2.1 Introduzione

Per machine learning, o apprendimento automatico, si intende una branca dell'intelligenza artificiale che si occupa di creare algoritmi e sistemi volti a migliorare le prestazioni di una macchina nel trovare determinati pattern, attraverso dati ed esperienze passate.
I modelli, nell'analisi dei dati, possono essere predittivi, quando cercano di prevedere risultati futuri e quindi ricavare informazioni su cosa potrebbe accadere o perché ciò accadrà, oppure descrittivi, quando si cercano correlazioni o sottogruppi di qualsiasi tipo nei dati conosciuti, cercando di ottenere informazioni utili quindi sul presente anziché sul futuro.
Il ruolo dell'informatica, in questo caso, si divide in due: in primo luogo quello di creare algoritmi efficienti per esaminare i numerosi dati e *allenare* l'intelligenza a risolvere il problema di ottimizzazione; in secondo luogo, una volta che il modello è stato assimilato, c'è bisogno di ottimizzare l'algoritmo che risolve l'inferenza, facendo in modo che anch'esso sia efficiente.
Le tecniche di apprendimento del machine learning si possono dividere principalmente in tre categorie: supervised, unsupervised e reinforcement:

## 2.2 Apprendimento supervisionato

Nel supervised learning sono forniti dataset di input e output già etichettati, quindi il sistema può *allenarsi* analizzando le correlazioni e verificando autonomamente l'esito delle sue previsioni. Il processo si divide principalmente in quattro fasi: viene preparato il dataset associando correttamente gli input, o *features*, con gli output, viene diviso il dataset in dati utilizzati per l'allenameno del modello e dati per testare il modello successivamente, viene allenato l'algoritmo sul set di training, quindi vengono apprese relazioni tra input e output e infine viene verificato quanto appreso sul set di test.
Gli algoritmi utilizzati per risolvere problemi mediante l'apprendimento supervisionato possono essere di due tipi principalmente: di classificazione e di regressione.
Con la classificazione si creano algoritmi capaci di assegnare i dati in categorie specifiche.

Quindi si può tradurre in una funzione che ha come variabili delle caratteristiche e in output produce un etichetta che identifichi l'insieme degli variabili. Un esempio è un modello che dato un dataset di immagini di paesaggi riesca a riconoscere se queste siano immagini di montagna, collina, mare o città. In questo caso le caratteristiche sarebbero ricavate automaticamente dal modello analizzando le immagini. Gli algoritmi di classificazione più diffusi sono: classificatore lineare, macchine a vettori di supporto, albero di decisione e foresta casuale.
Con la regressione si usa un algoritmo per prevedere un valore numerico a partire da una funzione che è composta dalle caratteristiche in input. Un esempio è un modello che riceve un dataset contenente caratteristiche di abitazioni, come per esempio metratura, numero di stanze, numero di piani, posizione, presenza di giardino, ecc., e che prevede il prezzo di queste case. Gli algoritmi di regressione più diffusi sono: regressione lineare, logistica, polinomiale, macchine a vettori di supporto, albero di decisione e foresta casuale.

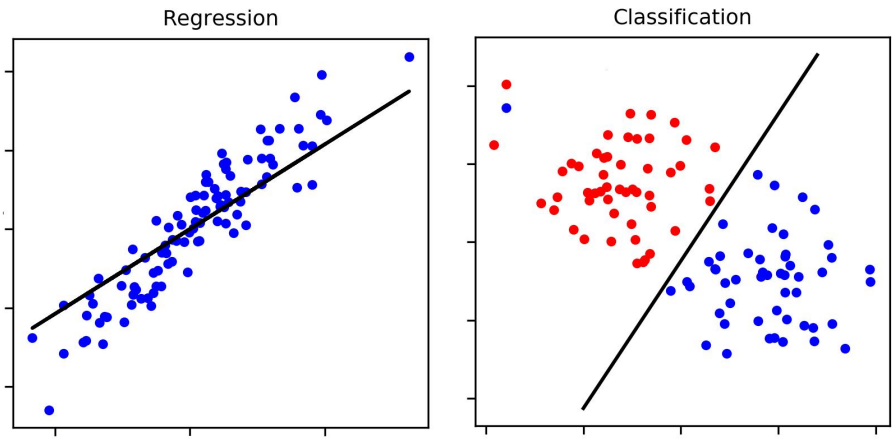



Figura 2.1: Esempi di funzione di regressione e classificazione

## 2.3 Apprendimento non supervisionato

L'unsupervised learning invece utilizza algoritmi per analizzare gruppi di dati non etichettati, quindi senza l'aiuto umano cercano di trovare schemi e correlazioni nel dataset. Gli algoritmi sono suddivisi in tre categorie principali: di clustering, associativi e di riduzione della dimensionalità.
Il clustering è una tecnica di data mining che si occupa di raggruppare i dati in base a delle similarità nelle loro caratteristiche, in modo da poter creare dei gruppi che possano contenere i dati non etichettati. La tecnica viene usata per gli algoritmi di compressione di immagini e documenti e anche per la segmentazione di clienti e del mercato, in quanto capace di trovare relazioni nascoste a causa della numerosa mole di dati o del loro essere implicite. Esempi di algoritmi sono: k-means, basati sulla densità e sulla distribuzione.

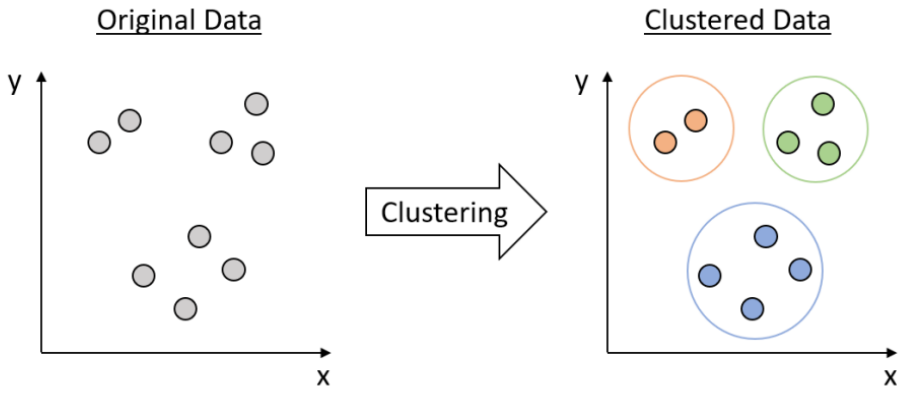



Figura 2.2: Clustering rappresentato graficamente

Gli algoritmi associativi usano diverse regole per trovare associazioni tra i dati, specialmente quelli che spesso compaiono insieme in grandi database, permettendo per esempio di eseguire analisi sui prodotti che vengono acquistati insieme (Market Basket Analysis) e sui sistemi di raccomandazione. Uno degli algoritmi di associazione più diffusi è apriori. Gli algoritmi di riduzione della dimensionalità vengono usati per diminuire la dimensione dei dataset in quanto troppo pesanti, estraendo le feature più importanti o ricombinandone altre, cercando di mantenere integrità dei dati. Vengono usati per preprocessare i dati come per esempio negli autoencoder per rimuovere elementi di disturbo dai dati.

## 2.4 Apprendimento semi-supervisionato

Esiste in realtà una quarta categoria non citata precedentemente, il semi-supervised learning. Come suggerisce il nome, è un ibrido delle tecniche precedenti. Con questi modelli si cerca di utilizzare una porzione di dati etichettata con una più consistente porzione di dati non etichettata, questo soprattutto quando sarebbe troppo dispendioso in termini di risorse etichettare tutto il dataset oppure quando si ha un incremento costante di nuovi dati. A partire dai dati *labeled*, il modello cerca di aggiungere tag ai dati sprovvisti. Il semi-supervised learning ha varie applicazioni come per esempio il riconoscimento vocale o la classificazione di contenuti web o testuali.

# Capitolo 3

# Apprendimento per Rinforzo

## 3.1 Introduzione

In casi come un modello che impari a vincere in un gioco, le tecniche di apprendimento supervisionato e non supervisionato non sono ottimali, per vari motivi. In determinati giochi non esiste sempre una mossa migliore, ma una successione di azioni che permettono di arrivare al risultato ottimo, e soprattutto non si riesce ad avere sempre un feedback istantaneo ma generalmente si ottiene solo alla fine di tutte le decisioni, sottoforma di vittoria o sconfitta. Guardando per esempio gli scacchi, mangiare un pezzo, che denota un risultato positivo istantaneo, non implica direttamente la vittoria della partita, che è il risultato atteso. Utilizzando il supervised learning in questo caso bisognerebbe insegnare al modello la mossa da eseguire in ogni situazione, che è impraticabile visto il numero di configurazioni possibili. Oppure in determinati ambienti non vi è una sconfitta vera e propria ma l'obiettivo potrebbe essere la riduzione del tempo di successo o minimizzare il numero di azioni. Non vi sarà mai una decisione errata ma alla fine del gioco si potrà scoprire il risultato ottenuto. La scelta più adatta per creare modelli che risolvano questi problemi è l'utilizzo di tecniche di apprendimento per rinforzo.

Il reinforcement learning è un metodo di apprendimento dove il modello impara effettuando delle azioni e interagendo con ciò che lo circonda. Con il passare del tempo l'intelligenza capisce quali azioni compiere in determinati momenti, utilizzando funzioni e algoritmi. Quindi non vi è un'influenza esterna che determina quale sia la migliore delle azioni ma vi è un miglioramento autonomo, partendo scelte aleatorie fino ad arrivare ad eseguire scelte più adatte secondo quanto stimato.
Per applicare tecniche di reinforcement learning è necessario che l'agente possa percepire lo stato dell'ambiente e che possa agire su di esso per modificarlo. Inoltre l'agente deve avere un obiettivo finale che si riferisca allo stato dell'ambiente. Una delle complessità che si presenta utilizzando l'apprendimento per rinforzo è quella che viene chiamata *exploration-exploitation tradeoff*. Per massimizzare il premio ottenuto l'agente preferisce *sfruttare* azioni che ha scoperto essere le più efficienti, ma al contrario per scoprire queste

azioni l'agente deve averle prima provate, ha quindi bisogno di *esplorare* randomicamente nuove decisioni. Chiaramente la decisione di svolgere unicamente una di queste azioni porterebbe al fallimento nella risoluzione del problema. Questo dilemma sta ancora venendo studiato da matematici in quanto ancora non si è giunti ad una risoluzione. Oltre all'esempio di un agente che impara a giocare, il reinforcement learning ha anche altre applicazioni come nella creazione di intelligenze per auto a guida autonoma o nell'elaborazione del linguaggio naturale.

Quando si parla di reinforcement learning, oltre ai già citati agente e ambiente, si possono identificare anche quattro sottoelementi, necessari per ottimizzare l'apprendimento: policy, reward signal, funzione valore e, se necessario, un modello per l'ambiente. La policy indica all'agente come comportarsi in un determinato istante, cioè traduce lo stato che riceve dall'ambiente in un'azione da compiere. Quindi una policy può variare da una semplice funzione a un problema di calcolo più complesso.
Il reward signal definisce l'obiettivo del problema. L'ambiente invia all'agente un premio (*reward*) sotto forma di numero, questo per ogni unità temporale del ciclo. L'obiettivo dell'agente è massimizzare il totale dei premi. Il reward signal permette quindi di decidere come modificare la policy, per fare in modo che possa si possa ottenere un reward più alto.
Tornando all'esempio degli scacchi, seguendo solo il reward signal si allenerebbe un'intelligenza a cercare di mangiare più pezzi possibili nell'immediato. Non si terrebbe quindi conto della vera condizione di vittoria che non è determinata solo dalle singole mosse. Per questo si utilizza anche la funzione valore, che permette di definire quali sono le azioni che portano ad un risultato migliore a lungo termine. Il valore di uno stato è la somma totale dei premi che l'agente prevede di ottenere nel corso del tempo, a partire da quello stato. Il valore indica quindi lo stato atteso desiderato, dopo aver preso in considerazione i vari stati previsti e il loro premio. Ci possono quindi essere stati con un premio molto basso ma che hanno un valore alto in quanto sono, molto probabilmente, susseguiti da stati con premio più alto. Sebbene il reward sia necessario per calcolare il valore, quest'ultimo è il dato che viene considerato per selezionare l'azione migliore da svolgere, in quanto ci permette di calcolare il premio maggiore nel lungo periodo. Chiaramente è più difficile da stimare in quanto deve essere calcolato costantemente osservando il susseguirsi degli stati, mentre il reward viene fornito dall'ambiente istantaneamente. Infatti uno degli obiettivi chiave quando si crea un algoritmo di reinforcement learning è quello di riuscire a creare una funzione che stimi efficientemente il valore.
L'ultimo elemento che viene utilizzato in alcuni sistemi è il modello dell'ambiente. Questi modelli imitano il comportamento dell'ambiente e vengono usati per fare previsioni su di esso. I modelli vengono quindi usati per pianificare la decisione a differenza dei sistemi *trial-and-error* che si basano sul fare tentativi e capire cosa ha funzionato e cosa no.

## 3.2 Caso a stato singolo: *K*-Armed Bandit

In questo esempio si considera un'ipotetica slot machine con $k$ leve. L'azione possibile è tirare una delle leve, e ad ognuna corrisponde una quantità di denaro in premio. L'obiettivo finale è massimizzare il denaro ottenuto da questa slot machine, dopo un determinato numero di tempo. Nel nostro esempio, per ognuna delle $k$ azioni, vi è un premio atteso. Chiamando l'azione selezionata $A_t$, per ogni istante $t$, e il premio corrispondente $R_t$, possiamo scrivere il valore $q_*(a)$ in funzione dell'azione $a$ come il premio atteso effettuando quest'ultima:

$$q_*(a) \doteq \mathbb{E}[R_t|A_t = a]$$

Chiaramente se si conosce il risultato certo di ogni azione la soluzione sarebbe triviale, in quanto basterebbe selezionare la leva che produce il premio maggiore. In questo esempio consideriamo quindi di avere una stima del risultato che chiameremo $Q_t(a)$. In ogni istante temporale vi sarà sempre uno di questi valori che sarà più alto degli altri. L'azione conseguente viene chiamata *greedy*. Utilizzando quest'ultima si *sfrutta* la conoscenza acquisita per ottenere un premio più alto. Mentre scegliendo una delle altre si *esplora*, permettendo di migliorare la stima di quelle che potrebbero essere azioni più vantaggiose. Esplorare quindi potrebbe portare a reward più alti nel lungo termine in quanto potrebbero esserci azioni con una stima incerta, che a seguito di numerosi cicli potrebbero rivelare un valore più alto della attuale azione *greedy*.

I metodi per stimare il valore di un azione vengono chiamati *action-values method*. In questo caso la stima migliore è la media dei valori ottenuti dopo aver svolto quest'azione. La media $Q_t(a)$ è inizializzata con un valore di default o a 0 quando il denominatore è uguale a 0. Il booleano $\mathbb{1}_{condizione}$ vale 1 se *condizione* è vera, 0 altrimenti.

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

Per selezionare l'azione da svolgere si può usare un metodo chiamato $\varepsilon - greedy$. L'algoritmo decide l'azione utilizzando un coefficiente $\varepsilon$, che permette di scegliere tra la mossa più *avida*, con una probabilità di $1-\varepsilon$ oppure una casuale, con una probabilità di $\varepsilon$ quindi.

Un altro metodo utilizzabile è l'Upper-Confidence-Bound Action Selection, che sceglie l'azione non avida che ha una potenzialità maggiore di essere ottima, osservando quanto precisamente è stata stimata e quanto manca affinché la stima sia massima. Una formula che si puo applicare è

$$A_t \doteq \arg\max_a \left[ Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

dove $N_t(a)$ denota il numero di azioni che sono state scelte fino al tempo $t$ e il numero $c > 0$ decide il grado di esplorazione. L'idea dell'UCB action selection è che il termine

sotto radice indichi il grado di incertezza del valore: quando il numero di volte in cui $a$ viene scelto aumenta, questo grado diminuisce in quanto $N_t(a)$ si trova al denominatore, mentre invece l'incertezza aumenta quando il tempo passa ma non viene scelta l'azione $a$, perché al denominatore vi è $t$; viene usato un logaritmo per ridurre l'incremento del numeratore col tempo.

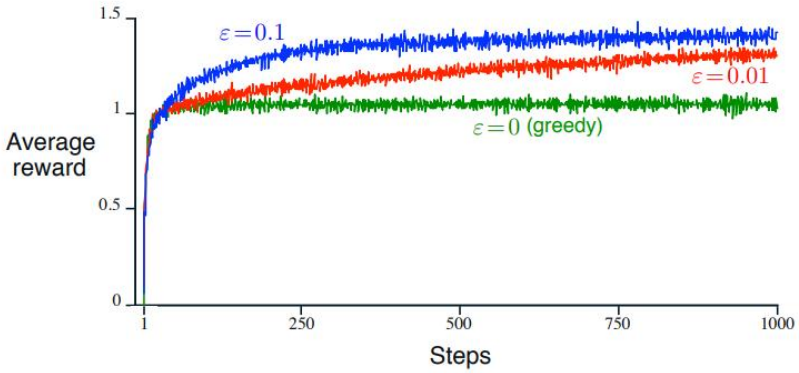



Figura 3.1: Risultati utilizzando metodo $\varepsilon - greedy$ con diverse $\varepsilon$

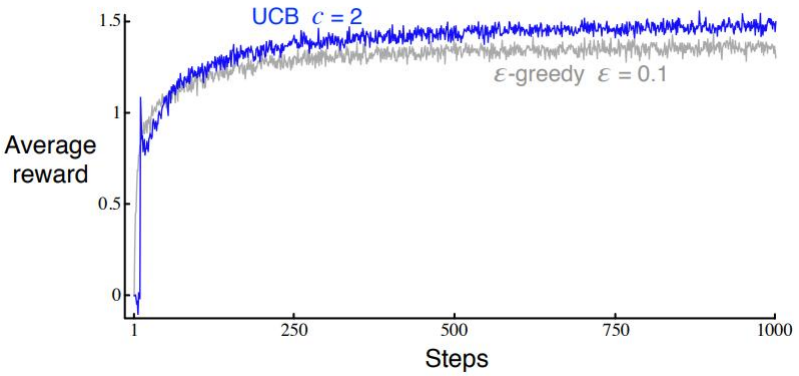



Figura 3.2: Confronto tra metodo UCB e $\varepsilon - greedy$

## 3.3 Processi Decisionali di Markov Finiti

I finite Markov decision processes (finite MDPs) sono un processo di decisione che valuta sia il feedback istantaneo, che gli aspetti associativi, scegliendo azioni diverse in situazioni differenti. I MDPs sono una formalizzazione classica della scelta sequenziale di decisione, dove le azioni infuenzano il premio immediato, gli stati sucessivi e i premi associati ad essi. Infatti quando stimiamo il valore $q$ utilizziamo una funzione $q(s, a)$ di ogni azione in ogni stato oppure il valore $v(s)$ di ogni stato selezionando l'azione ottima. I MDPs mirano ad essere una struttura standard nei problemi di'apprendimento mediante interazione, in cui si cerca di raggiungere un obiettivo. L'intelligenza che impara e decide le azioni da eseguire viene chiamata *agente* e ciò con cui interagisce è l'*ambiente*. Questi

due sono in continuo scambio, il primo selezionando le azioni mentre il secondo restituendo nuove situazioni, premi e valori che l'agente cerca di massimizzare. Lo scambio di informazioni avviene in una sequenza di passi temporali discreti. Ad ogni tempo $t$ l'agente riceve uno stato $S_t \in \mathbb{S}$, in base a questo decide un azione $A_t \in \mathbb{A}(s)$. L'istante successivo l'agente riceve un premio $R_{t+1} \in \mathbb{R} \subset \mathbb{R}$ e il nuovo stato $S_{t+1}$.

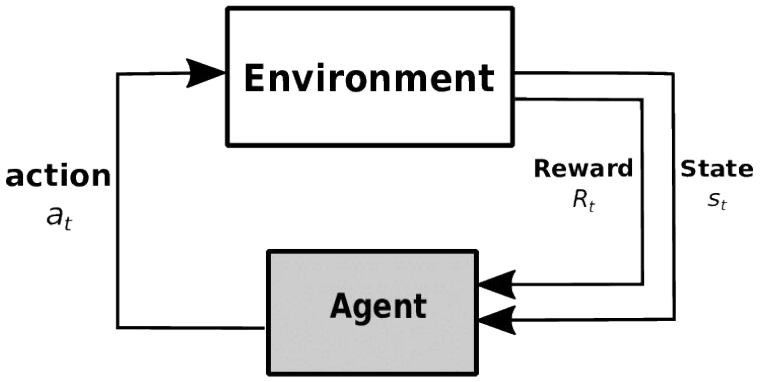



Figura 3.3: Schema interazione tra agente ed ambiente

Nei MDP finiti i set di stati, azioni e reward hanno un numero finito di elementi. In questo caso le variabili aleatorie $R_t$ e $S_t$ hanno una distribuzione di probabilità discreta che dipende solo dallo stato e azione precedenti. Prendendo due valori di queste variabili aleatorie, $s'$ e $r$ c'è una funzione di probabilità che questi valori si presentino in un tempo $t$ dati un preciso valore di stato e azione precedenti:

$$p(s', r|s, a) \doteq Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

In un processo di decisione di Markov, le probabilità fornite da $p$ caratterizzano completamente la dinamica dell'ambiente e dipendono unicamente dallo stato e azione immediatamente precedenti, e non da tutti quelli passati. Questa è vista come una restrizione sullo stato, che deve includere le informazioni dell'interazione passata. Se questa condizione viene rispettata allora il modello rispetta la *Proprietà di Markov*.

L'obiettivo definito precedentemente per un algoritmo di reinforcement learning è quello di massimizzare il premio a lungo termine, chiamato ritorno atteso $G_t$. Questo può essere inteso in casi molto semplici come la somma di tutti i reward ottenuti fino ad allora. Questo approccio ha senso quando applicato in situazioni in cui vi è un naturale decorrere di azioni, senza una interruzione apparente. Vengono chiamate task continue e tra gli esempi c'è quello di robot oppure di problemi di controllo automatico. Invece vi sono casi in cui l'interazione agente-ambiente si interrompe e suddivide naturalmente naturalmente in vari segmenti temporali finiti chiamati episodi. Ogni episodio finisce in uno stato di nome terminale ed è generalmente seguito da un reset ad uno stato iniziale. Esempi immediati di task episodiche, così sono chiamate, sono giochi strutturati in partite. Trovare una funzione di ritorno per le task continue è problematico in quanto il segmento temporale finale sarebbe a $T = +\infty$ e quindi potrebbe divergere, ritrovandosi

a massimizzare un valore infinito. Quindi si utilizza un valore $\gamma$, con $0 \leq \gamma \leq 1$, chiamato *discount rate* che permette di calcolare il ritorno atteso *scontato*, e quindi utilizzare il massimo per scegliere l'azione. Per i casi più semplici si può usare nuovamente una somma:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \lambda^k R_{t+k+1}$$

Il discount rate dà un peso ai premi attesi, più lontano sono nel tempo e meno influenza hanno. Quando ha un valore minimo si priorizzerà il premio immediato mentre avvicinandosi a $\gamma = 1$ veranno considerati di più anche i reward futuri. Con un raccoglimento di $\gamma$ si può semplificare la formula evidenziando la correlazione tra segmenti temporali successivi:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Come specificato in precedenza, una policy è una mappatura tra stato e probabilita di scelta di una determinata azione. Se l'agente sta seguendo una policy $\pi$ allora $\pi(a|s)$ è la probabilità che venga scelta l'azione $a$ quando si riceve uno stato $s$, in quanto $a$ e $s$ appartengono a $A_t$ e $S_t$ che sono distribuzioni di probabilità. Per i MDPs possiamo definire una funzione valore $v_\pi(s)$ e una funzione stato-valore $q_\pi(s, a)$ che sono rispettivamente il ritorno atteso quando si comincia in uno stato $s$ e poi si segue la policy $\pi$ e quando avviene la medesima cosa ma tenendo anche conto dell'azione $a$ svolta.

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}[\sum_{k=0}^{\infty} \lambda^k R_{t+k+1} | S_t = s, A_t = a] \end{aligned}$$

Grazie alla ricorsività di queste formule possiamo ricavare quella che viene chiamata l'equazione di Bellman di $v_\pi$ che esprime una relazione il valore dello stato attuale e successivo. Con queta semplificazione la formula di si riduce ad una somma sulle tre variabili, dove $s'$ è lo stato seguente.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi \left[ G_t \mid S_t = s \right] \\ &= \mathbb{E}_\pi \left[ R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \left[ r + \gamma \mathbb{E}_\pi \left[ G_{t+1} \mid S_{t+1} = s' \right] \right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p\left(s', r \mid s, a\right) \left[ r + \gamma v_\pi\left(s'\right) \right] \end{aligned}$$

Nei MDPs scegliere una policy ottima è molto semplice. Basta selezionare la $\pi_*$ che restituisce il valore più alto (possono essere anche multiple in caso di risultati uguali). Queste condividono la stessa funzione stato-valore $v_*(s)$ e la stessa funzione azione-valore

$q_*(s, a)$, entrambe ottime. Quest'ultima, a partire dalla coppia stato-azione, restituisce il ritorno atteso per aver compiuto l'azione $a$ nello stato $s$ e aver seguito la policy ottima. Quindi si può riscrivere $q_*$ in funzione di $v_*$:

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

Anche le funzioni ottime devono rispettare il principio di autoconsistenza dell'equazione di Bellman, essendo però un valore ottimo, si può ignorare il riferimento ad una policy specifica, arrivando a trovare questa *equazione di Bellman ottima* di $v_*$:

$$v_*(s) = \max_a \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma v_\pi\left(s'\right)\right]$$

L'equazione di Bellman ottima di $q_*$ è invece:

$$q_*(s, a) = \sum_{s',r} p\left(s', r \mid s, a\right)\left[r + \gamma \max_a q_*\left(s', a'\right)\right]$$

Per i MDPs finiti l'equazione ottima di Bellman ha una soluzione unica. In realta questa equazione è un sistema di $n$ equazioni, una per ogni stato, ma se si conosce la dinamica $p$ dell'ambiente si riesce a risolvere il sistema con i metodi classici. Per ognuno degli stati ci saranno una o più azioni che massimizzano il valore di questo sistema, e sono tutte azioni ottime. La comodità di $v_*$ è che utilizzando un approccio greedy, quindi massimizzando solo il premio immediato, abbiamo comunque un risultato che tiene in considerazione anche gli stati a lungo termine. Purtroppo, per molte applicazioni, questo approccio e difficilmente utile, in quanto calcolare l'equazione per ogni stato sarebbe un compito che utilizzerebbe un tempo troppo elevato persino con i calcolatori più potenti, spesso le dinamiche dell'ambiente non sono note precisamente e non tutti gli stati rispettano la proprietà di Markov. Quindi tipicamente ci si affida a dei calcoli approssimativi dell'equazione di Bellman.

## 3.4 Cenni dell'apprendimento mediante differenza temporale

Il temporal-difference learning (TD) è uno dei metodi centrali del reinforcement learning. Combina idee provenienti dall'approccio Monte Carlo e dalla programmazione dinamica (DP). Con il primo condivide l'apprendimento diretto tramite esperienza, senza la conoscenza della dinamica dell'ambiente, mentre con il secondo l'aggiornamento delle stime, ad ogni step temporale, senza attendere il risultato finale (quindi fanno *bootstrap*. La stima $V$ di $v_\pi$ più semplice per un metodo TD è:

$$V\left(S_t\right) \leftarrow V\left(S_t\right) + \alpha\left[R_{t+1} + \gamma V\left(S_{t+1}\right) - V\left(S_t\right)\right]$$

Trovata applicando una formula di aggiornamento standard: $NuovaStima \leftarrow VecchiaSti-ma + DimensioneStep[Obiettivo - VecchiaStima]$.

Un'algoritmo di controllo TD può essere *on-policy* oppure *off-policy* a seconda che, aggiornando il valore, tenga in considerazione un azione predefinita oppure un azione che segua la policy attuale. Nel 1989 venne creato *Q-learning*, un algoritmo di controllo TD off-policy, definito da questa formula:

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

Il valore $Q$, in questo caso, approssima direttamente la funzione ottima azione-valore, indipendentemente dalla policy seguita, semplificando drasticamente l'analisi dell'algoritmo e permettendo di dimostrare facilmente la convergenza.

# Capitolo 4

# Snake

## 4.1 Introduzione alle reti neurali

Le reti neurali artificali (o artificial neural networks) sono formate da neuroni e connessioni e sono tipicamente organizzate in layers. Di questi si distinguono l'input e l'output, che contengono rispettivamente i dati conosciuti che vengono assegnati alla rete, e il risultato che calcolo che esegue la rete. Tra questi due layer vi possono essere uno o più *hidden layers*. I neuroni agiscono come funzioni che processano il loro segnale come una combinazione pesata degli input ricevuti, producendo un segnale in output. Questa funzione è chiamata funzione di attivazione, una delle più popolari è l'unità lineare rettificata (ReLU). I neuroni sono collegati con pesi $w$ e per ogni neurone $j$ vengono sommati i pesi arrivati in input e poi processati dalla funzione d'attivazione $\sigma$. L'output $o$ è quindi $o_j = \sigma(\sum_i x_i w_{ij})$, considerando i pesi $ij$ del neurone precedente $x_i$. L'output viene quindi inviato in input ai neuroni successivi.

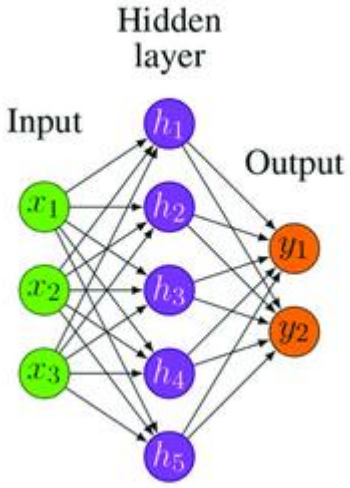



Figura 4.1: Schema di una Rete Neurale

Le reti neurali sono funzioni parametriche e allenare la rete implica l'aggiustare i pesi (i parametri), per trovare una relazione input-output desiderata. Per arrivare a questo obiettivo bisogna minimizzare la funzione di errore (o *loss function*) che calcola la diffe-

renza tra l'output ottenuto e quello atteso. Per svolgere determinati compiti, una rete neurale con layer singolo non sarebbe abbastanza efficiente, come per esempio per il riconoscimento di immagini, e quindi vengono utilizzate quelle che sono chiamate *multi-layer neural networks*, quindi reti con molteplici hidden layers.

## 4.2 IA per Snake

Snake è un videogioco creato nel 1977 in cui si controlla un serpente in un ambiente bidimensionale con visuale dall'alto. Il serpente si muove automaticamente nella direzione in cui sta guardando e il giocatore può decidere di cambiarla. All'interno del campo vi è anche una mela piazzata casualmente, e quando il serpente entra in contatto con essa, lui cresce di dimensioni e ne viene piazzata un'altra all'interno dell'ambiente. L'obiettivo del giocatore è quello di riuscire a far crescere il serpente fino alla dimensione massima possibile (dettata dall'interfaccia). Il giocatore perde la partita quando il serpente entra in contatto con i bordi dello schermo oppure con un pezzo della sua coda.

Per creare un programma che applicasse il reinforcement learning e le reti neurali per imparare a giocare a Snake ho scelto di utilizzare il linguaggio di programmazione Python. La base di gioco che ho utilizzato è una versione di Snake creata con pygame. Come prima cosa ho deciso i valori dei premi: +10 se viene mangiata una mela, -10 se il serpente si mangia la coda o collide con i bordi e 0 per ogni azione non risultante in nessuna delle due condizioni precedenti.

```
if self.is_collision() or self.frame_iteration > 400*len(self.snake):
    game_over = True
    reward = -10
    return reward, game_over, self.score
elif self.frame_iteration > 200*len(self.snake):
    reward = -5
```

Ho deciso anche di interrompere il gioco e considerare la partita come persa in caso l'iterazione di gioco per ottenere un punto superasse di 400 volte la lunghezza del serpente, e di applicare un reward di -5 appena questa lo superasse di 200 volte. Questo per scoraggiare le tecniche, che il modello ogni tanto applicava, di loop continuo per mantenere un reward stabile ed evitare reward negativi.

Le azioni che il serpente può eseguire sono tre: decidere se continuare in una direzione, girare a sinistra o girare a destra.

```
def _move(self, action):
    clock_wise = [Direction.RIGHT, Direction.DOWN, Direction.LEFT,
   Direction.UP]
    idx = clock_wise.index(self.direction)

    if np.array_equal(action, [1, 0, 0]):
```

```
        new_dir = clock_wise[idx]
    elif np.array_equal(action, [0, 1, 0]):
        next_idx = (idx + 1) % 4
        new_dir = clock_wise[next_idx]
    else:
        next_idx = (idx - 1) % 4
        new_dir = clock_wise[next_idx]

    self.direction = new_dir
```

In base all'azione ricevuta, un vettore di tre elementi in cui uno solo ha il valore indicante la decisione non nullo, viene aggiornata la direzione del serpente. Viene utilizzato un indice incrementale ciclico per dare un senso ordinato alle direzioni, in questo caso orario. Queste sono le modifiche principali apportate al gioco.

L'agente è composto da una funzione che lo inizializza, utilizzando un *learning rate* di $lr = 0.001$, un *discountrate* di $\gamma = 0.9$ ed una memoria massima di 100 000 elementi, funzioni accessorie e la funzione *train()* che è anche il main del programma.

```
def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = SnakeGameAI()
    while True:
        state_old = agent.get_state(game)

        final_move = agent.get_action(state_old)

        reward, done, score = game.play_step(final_move)
        state_new = agent.get_state(game)

        agent.train_short_memory(state_old, final_move, reward,
    state_new, done)

        agent.remember(state_old, final_move, reward, state_new, done)

        if done:
            game.reset()
            agent.n_games += 1
            agent.train_long_memory()

            if score > record:
                record = score
                agent.model.save()
```

Il ciclo dell'agente è il seguente: richiede lo stato attuale dal gioco, calcola la mossa migliore da compiere, la esegue, ricevendo premio, punteggio e un boolean per confermare che la partita non sia finita, infine richiede il nuovo stato. Per ognuno degli step temporali di una partita allena il modello a breve termine e memorizza le informazioni dello stato precedente e attuale. Quando la partita finisce utilizza tutte queste informazioni per allenare la memoria a lungo termine e fa ripartire il gioco. In caso di nuovo record si salva il modello.
Come stati ho scelto 15 variabili che hanno un valore booleano e sono 3 che indicano se a sinistra, davanti o all'immediata destra del serpente vi è un pericolo (confini del gioco oppure la coda), 4 che indicano la direzione cardinale in cui il serpente sta andando (ex: 1,0,0,0 = sinistra), 4 che indicano la posizione della mela rispetto alla testa (ex: 1,0,1,0 = in alto a sinistra) e infine le ultime 4 che allo stesso modo della mela indicano la posizione dell'ultimo blocco della coda. Chiamando la funzione get_state(self,game) viene restituita quindi un array di 15 elementi.

```
def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

def train_long_memory(self):
    if len(self.memory) > BATCH_SIZE:
        mini_sample = random.sample(self.memory, BATCH_SIZE)
    else:
        mini_sample = self.memory

    states, actions, rewards, next_states, dones = zip(*mini_sample)
   #estraggo le diverse variabili
    self.trainer.train_step(states, actions, rewards, next_states, dones
   )

def train_short_memory(self, state, action, reward, next_state, done):
    self.trainer.train_step(state, action, reward, next_state, done)
```

Per memorizzare le informazioni passate utilizzo un deque, in quanto appena raggiunta la memoria massima essa rimuoverà automaticamente gli elementi più vecchi. Per allenare la memoria a lungo termine utilizzo una BATCH_SIZE=1000 scelta arbitrariamente. In caso il numero di elementi nella memoria superasse la BATCH_SIZE allora li seleziono casualmente. La memoria a corto termine e a lungo termine vengono allenate allo stesso modo, la prima però passa solo un valore per tipo e lo fa ad ogni step temporale di gioco, mentre la seconda passa numerose variabili per tipo, ma alla fine di ogni partita.

```
def get_action(self, state):
    self.epsilon = 80 - self.n_games
    final_move = [0,0,0]
    if random.randint(0, 200) < self.epsilon:
```

```
        move = random.randint(0, 2)
        final_move[move] = 1
    else:
        state0 = torch.tensor(state, dtype=torch.float)
        prediction = self.model(state0)
        move = torch.argmax(prediction).item()
        final_move[move] = 1

    return final_move
```

L'ultima funzione della classe Agent è get_action(self,state). Facendo vari test noto che intorno alle 100 partite la strategia del modello è molto solida, quindi utilizzando la variabile $\varepsilon$ aggiungo un tasso di randomizzazione dell'azione che arriva ad essere nullo dall'$80^a$ partita. Questo è il metodo che ho usato per bilanciare l'*exploration-exploitation tradeoff*. In caso non venisse randomizzata la mossa, la faccio prevedere al modello passandogli lo stato attuale (convertito a tensore di float) e ne ricevo uno di 3 valori. Essendo float e non più booleani, considererò come 1 la posizione in cui si trova il valore più alto, e 0 le altre.

Per creare il modello ho utilizzato pytorch un framework di apprendimento automatico basata sulla libreria torch. Il modello è composto da due classi la classe Linear_QNet che è la nostra rete neurale e la classe QTrainer che si occupa di allenare l'intelligenza.

```
MAX_MEMORY = 100_000
BATCH_SIZE = 1000
LR = 0.001

class Agent:
    def __init__(self):
        self.n_games = 0
        self.epsilon = 0
        self.gamma = 0.9
        self.memory = deque(maxlen=MAX_MEMORY)
        self.model = Linear_QNet(15, 512, 3)
        self.trainer = QTrainer(self.model, lr=LR, gamma=self.gamma)
    # Questa sezione e' presa dal Agent.py
```

```
class Linear_QNet(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = self.linear2(x)
        return x

```

```
12      def save(self, file_name='model.pth'):
13          model_folder_path = './model'
14          if not os.path.exists(model_folder_path):
15              os.makedirs(model_folder_path)
16
17          file_name = os.path.join(model_folder_path, file_name)
18          torch.save(self.state_dict(), file_name)
```

Per la rete neurale oltre ai 15 input e 3 output ho scelto di applicare un solo hidden layer, in quanto sufficiente per l'obiettivo del programma, di 512 elementi. Inizializzando la classe ottengo i due linear che sono delle reti a layer singolo disegnate per calcolare una funzione lineare $y = w*x$ dove $y$ è l'output, $x$ l'input e $w$ il peso. Utilizzo poi una funzione forward, che riceve come parametro il tensore $x$ con lo stato, per utilizzare la funzione di attuazione, in questo caso ReLU. Per il secondo layer non è necessaria in quanto uso direttamente i numeri che ottengo in output. Vi è poi una funzione per salvare il modello attuale in una cartella.

```
1  class QTrainer:
2      def __init__(self, model, lr, gamma):
3          self.lr = lr
4          self.gamma = gamma
5          self.model = model
6          self.optimizer = optim.Adam(model.parameters(), lr=self.lr)
7          self.criterion = nn.MSELoss()
8
9      def train_step(self, state, action, reward, next_state, done):
10         state = torch.tensor(state, dtype=torch.float)
11         next_state = torch.tensor(next_state, dtype=torch.float)
12         action = torch.tensor(action, dtype=torch.long)
13         reward = torch.tensor(reward, dtype=torch.float)
14
15         if len(state.shape) == 1:
16             state = torch.unsqueeze(state, 0)
17             next_state = torch.unsqueeze(next_state, 0)
18             action = torch.unsqueeze(action, 0)
19             reward = torch.unsqueeze(reward, 0)
20             done = (done, )
21
22         pred = self.model(state)
23         target = pred.clone()
24         for idx in range(len(done)):
25             Q_new = reward[idx]
26             if not done[idx]:
27                 Q_new = reward[idx] + self.gamma * torch.max(self.model(
    next_state[idx]))
28
29             target[idx][torch.argmax(action[idx]).item()] = Q_new
```

```
30
31          self.optimizer.zero_grad()
32          loss = self.criterion(target, pred)
33          loss.backward()
34
35          self.optimizer.step()
```

Nella classe QTrainer vengono inizializzati anche un ottimizzatore, in questo caso di tipo Adam e una loss function che in questo caso è l'errore quadratico medio. Nella funzione di allenamento converto tutti i dati in tensori e quelli che hanno dimensione 1 li riformatto in modo che siano (1, x). Implemento quindi la funzione Q, applicando l'equazione di Bellman per TD definita in precedenza:

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

Questa può essere riassunta in due formule:

$$Q = model.predict(state_{old}) \text{ e } Q_{new} = R + \gamma * max(Q(state))$$

che sono riga 22, per la prima, e 25 e 27, per la seconda, del codice. Quando la partita non è ancora finita il nuovo valore di Q sarà semplicemente il premio, quando finisco la partita aggiorno la Q con la stima. Dato che la previsione è un tensore di tre elementi e Q_new invece è un numero singolo, utilizzo target, che è un clone della previsione, e lo sovrascrivo con il nuovo valore alla posizione esatta.

## 4.3 Risultati

Con questi dati e questa struttura il modello comincerà ad avere una strategia solida dopo 80 partite. Riuscendo a puntare alla mela e ad evitare di andare direttamente incontro ad un ostacolo. Vi è però una criticità. Il modello non impara a non intrappolarsi con la sua stessa coda in quanto non possiede le informazioni per farlo. Infatti la maggior parte delle partite terminano perché il serpente si chiude su se stesso e rimane senza via di fuga. Un modo per poter superare questo problema sarebbe l'utilizzo di reti neurali convoluzionali e il passaggio di un maggior numero di dati, come per esempio l'intera griglia di gioco. Questo aumenterebbe la complessità del modello e richiedere un maggior numero di iterazioni per poterla allenare.

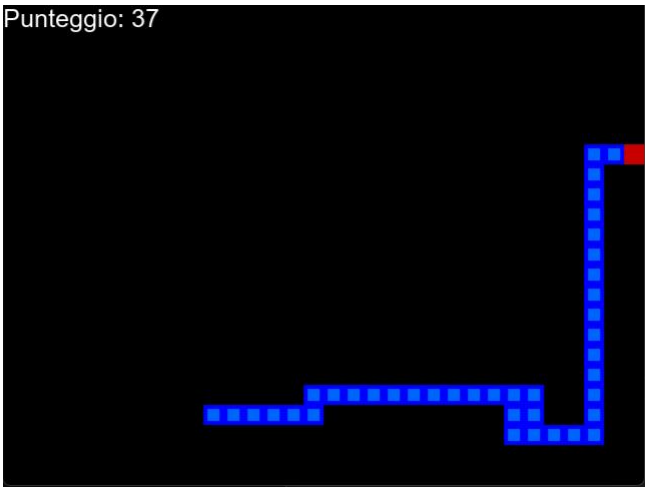
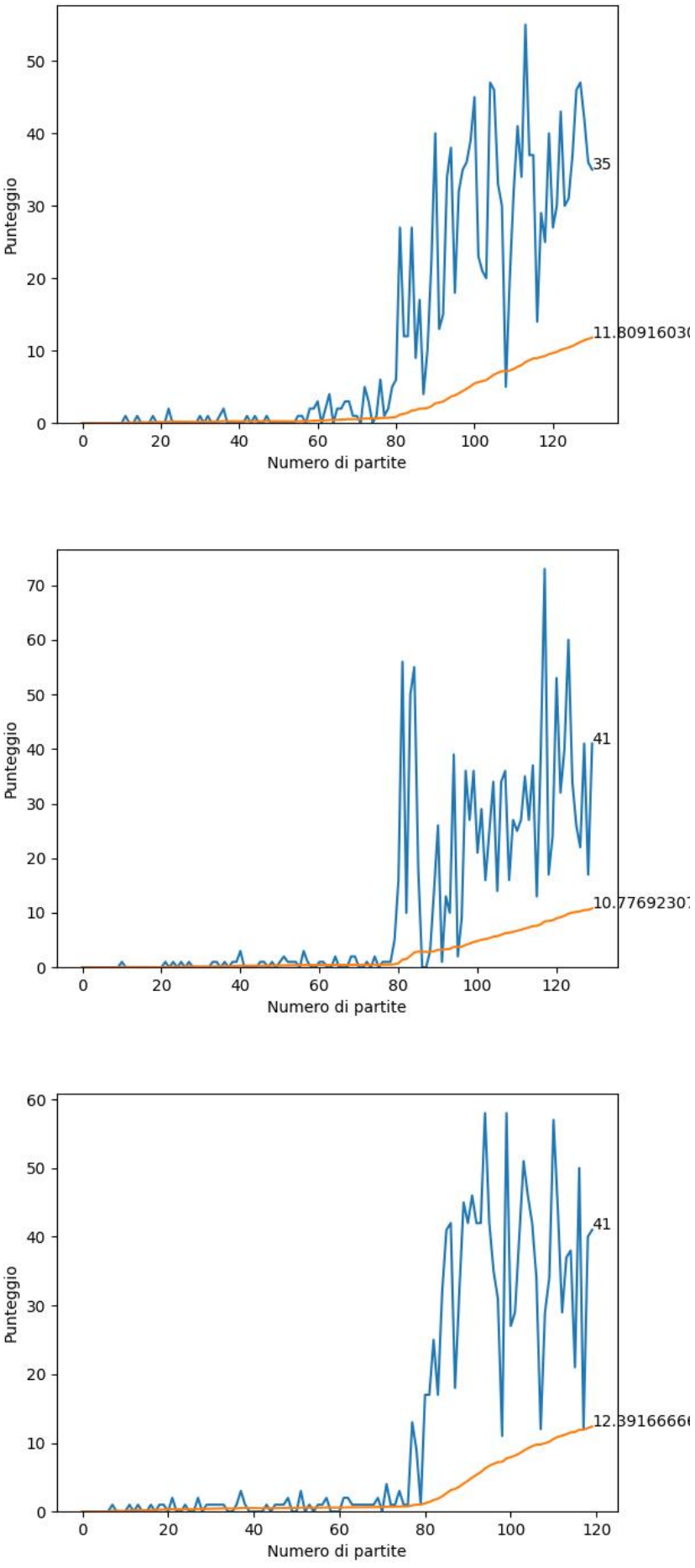



Figura 4.2: Schermata di gioco

Figura 4.3: Risultati di tre simulazioni



# Capitolo 5

# Conclusioni

Le potenzialità dell'apprendimento per rinforzo sono elevate. Negli ultimi anni si sono riusciti a fare progressi importanti, tanto che sono state create intelligenze artificiali capaci di battere campioni mondiali di giochi come Go e Scacchi, sebbene questi giochi richiedano una grandissima abilità nel valutare lo stato di gioco e pianificare le proprie mosse, prevedendo quelle dell'avversario. Il vantaggio di questi giochi per un IA è che gli stati non dipendono completamente dal caso e sono tutti prevedibili. Questo permette ai calcolatori di trovare mosse che un giocatore umano difficilmente contemplerebbe perché il beneficio che potrebbe portare sarebbe troppo distante per essere previsto. Per questa tesi sono riuscito a creare un intelligenza che impara autonomamente a giocare discretamente bene a Snake, senza aver bisogno di strumenti avanzati ma semplicemente con un personal computer. Ciò mi fa comprendere quanto questi strumenti possano essere potenti e quanto possano essere utilizzati e integrati col tempo nella vita quotidiana. Però, per esempio, fino a che punto è giusto automatizzare sempre di più mansioni, sostituendo persone con robot? Questa è una delle tante domande che mi pongo. È importante, secondo me, anche interrogarsi sull'etica dell'utilizzo di queste tecnologie.

# Bibliografia


[1] R. S. Sutton e A. G. Barto. *Reinforcement learning: An introduction*. 2ª ed. MIT Press, 2018.

[2] Tom Michael Mitchell. *Machine Learning*. Vol. 1. Lecture Notes in Mathematics. New York: McGraw-hill, 2007.

[3] Ethem Alpaydin. *Introduction to machine learning*. 3ª ed. MIT Press, 2020.

[4] S. J. Russell e P. Norvig. *Artificial intelligence a modern approach*. 3ª ed. Pearson Education, Inc., 2010.

[5] Aske Plaat. *Learning to play: reinforcement learning and games*. Springer Nature, 2020.

[6] Alan Mathison Turing. "COMPUTING MACHINERY AND INTELLIGENCE". In: *The Computer Journal* LIX.236 (October 1950), pp. 433–460. DOI: https://doi.org/10.1093/mind/LIX.236.433.

[7] Leo Gugerty. "Newell and Simon's logic theorist: Historical background and impact on cognitive modeling". In: *SAGE Publications* 50.9 (2006). DOI: https://doi.org/10.1177/154193120605000904.

[8] J. McCarthy et al. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. 1955. DOI: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.

[9] Saul McLeod. "Bf skinner: Operant conditioning". In: (2007). DOI: https://www.simplypsychology.org/operant-conditioning.html.