

Progetto Fondamenti di Intelligenza Artificiale

Realizzato da Dalmonte Alan

Reti Neurali applicate ai giochi da tavolo

Introduzione:

Il gioco da tavolo che ho scelto come base del progetto è Othello. Viene giocato da due persone, su una scacchiera di 8 caselle per lato, con 64 pedine bicolori. Un giocatore usa le pedine nere, l'avversario quelle bianche. La disposizione iniziale delle pedine è come in Figura 1. Inizia a giocare il Nero.

Al suo turno ogni giocatore poggia una pedina, con la faccia del proprio colore rivolta verso l'alto, su una casella ancora vuota (se uno dei due giocatori resta senza pedine, può usare quelle dell'avversario). Una pedina imprigiona quelle avversarie in una o più direzioni (orizzontale, verticale e/o diagonale), rendendo le pedine imprigionate del proprio colore (ovvero capovolgendole).

Il giocatore, al suo turno, è obbligato a giocare appoggiando una pedina in maniera da imprigionare almeno un disco avversario; non può porre una pedina in una casella senza girare dischi avversari, né girare meno di quelle richieste, né rinunciare alla mossa. Nel caso in cui non vi siano mosse legali, il giocatore passa, e tocca nuovamente all'avversario, fino all'esaurimento delle mosse per entrambi i giocatori (in genere ciò avviene dopo aver riempito interamente di pedine la scacchiera). Raramente può succedere che tutte le pedine diventino di un solo colore o che entrambi i giocatori non possano muovere anche se ci sono ancora caselle vuote, in tal caso la partita termina.

Vince chi, quando è stata giocata l'ultima mossa, ha più pedine dell'avversario. In caso di pari pedine, la partita è dichiarata patta.

Se capita di imprigionare più pedine, la fila da capovolgere deve essere continua e formata da dischi dello stesso colore. Nella figura 2 (e 3) il disco A è già sulla scacchiera. Giocando in B si imprigionano i tre dischi neri tra i due dischi bianchi.



Figura 1: Mossa al Nero

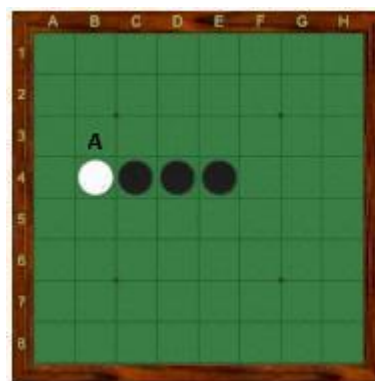


Figura 2: Mossa al Bianco

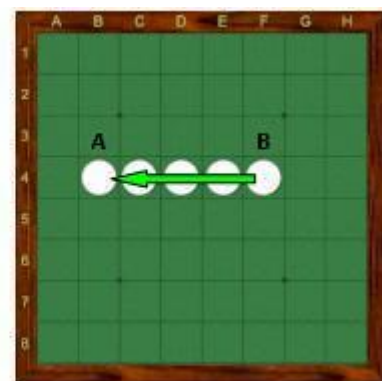


Figura 3: Mossa al Bianco

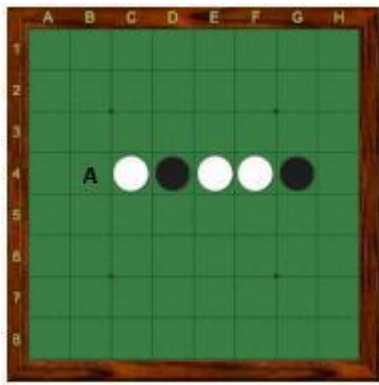


Figura 4: Mossa al Nero



Figura 5: Mossa al Nero

Nel caso seguente invece (Figura 4), il Nero, giocando in A, può capovolgere solo un disco (quello nella casella c4). In quest'altro caso, mostrato nella Figura 5, il Nero non può giocare in A perché non verrebbe girata alcuna pedina.

In Figura 8 il Nero può giocare in c5. Con quella mossa imprigionerà ben 9 pedine, in 5 direzioni diverse, raggiungendo la posizione di Figura 9.

Notiamo che la pedina in f5 non è diventata nera: è stata girata la pedina in d5 perché imprigionata fra c5 ed e5, ma f5, essendo oltre e5 stessa, non deve essere girata nonostante la presenza di g5 nera.

In questa posizione vi sono pedine bianche imprigionate fra altre pedine nere, ma non contano: diventano nere solo quelle imprigionate direttamente fra la pedina appoggiata ed un'altra dello stesso colore in una delle otto direzioni.



Figura 8: Mossa al Nero



Figura 9: Mossa al Nero

Struttura del progetto:

Il progetto è stato sviluppato con Python 3 e per la piattaforma di gioco mi sono rifatto ad un tutorial online effettuando alcuni cambiamenti e miglioramenti in corso d'opera (^[1]).

Le varie caselle della scacchiera sono rappresentate da un oggetto 'Square'; lo stato della casella è rappresentato dal valore della proprietà 'value' che risulterà None se la casella è vuota, 0 se è occupata da una pedina nera o 1 se la pedina è bianca. Inoltre, contiene altre variabili che verranno utilizzate per l'aggiornamento e la modifica della scacchiera.

La scacchiera completa, le mosse e i turni sono gestiti tutti dalla classe 'Board'. La classe si occupa di tutte le funzioni di gestione fra cui le principali sono: inizializzazione della scacchiera, gestione delle mosse valide, gestione delle pedine da capovolgere dopo una mossa effettuata.

La classe permette di ricevere mosse in due formati differenti, il primo prende delle coordinate numpy (x,y) e serve per semplificare l'utilizzo e la gestione delle mosse da parte di un giocatore automatico, mentre il secondo è alfanumerico ed è predisposto verso un giocatore umano, che è semplificato nella visione delle caselle seguendo la scacchiera, e quindi accetta input come 'A4'.

Un'altra funzione si occupa di disegnare la scacchiera in una maniera facilmente comprensibile alla visione di un giocatore umano; in particolare usa i codici unicode u25CF (●), u25CB (○), e u25E6 (◦) per rappresentare le caselle nere, bianche o vuote.

In [3]: b = Board()

1	◦	◦	◦	◦	◦	◦	◦	◦
2	◦	◦	◦	◦	◦	◦	◦	◦
3	◦	◦	◦	◦	◦	◦	◦	◦
4	◦	◦	◦	○	●	◦	◦	◦
5	◦	◦	◦	●	○	◦	◦	◦
6	◦	◦	◦	◦	◦	◦	◦	◦
7	◦	◦	◦	◦	◦	◦	◦	◦
8	◦	◦	◦	◦	◦	◦	◦	◦
	A	B	C	D	E	F	G	H

La piattaforma presenta anche una ulteriore classe chiamata 'AI' che rappresenta un giocatore automatico molto semplice di Othello. Questo giocatore è programmabile solamente per giocatore con mosse casuali scelte fra le mosse valide in quel turno della partita. Quindi non è presente alcuna euristica per questo giocatore; ma come riportato su alcuni articoli che trattano questo gioco la scelta della mossa in maniera casuale risulta essere una tecnica abbastanza efficace. Il motivo è dato dal fatto che anche una singola mossa può portare grandi benefici, come visto in figura 8, quindi la pianificazione sulle mosse avversarie risulta essere importante e un giocatore casuale distrugge molte strategie e alla fine potrebbe prevalere.

Reinforcement Learning:

Per questo progetto ho deciso di realizzare un giocatore che imparasse a giocare ad Othello utilizzando le tecniche di Reinforcement Learning. Queste tecniche si prestano molto bene a lavorare con giochi da tavolo, infatti risultano essere ideali per situazioni in cui si presentano dei benefici a lungo termine, come una mossa che può portare uno sfavore nel breve termine ma assicurarmi una mossa molto favorevole più avanti nella partita.

Per realizzare il giocatore ho deciso di implementare una rete neurale, principalmente realizzata tramite Tensorflow e Keras, e l'algoritmo del Q-learning.

L'algoritmo 'Q-learning' si basa su una funzione $Q(s,a)$ che, per un dato stato s e una data azione a , restituisce una stima di una ricompensa totale che avremmo ottenuto partendo da questo stato e effettuando quell'azione. Il nostro scopo sarebbe trovare per ogni stato la $\text{argmax}_a Q(s,a)$, cioè sapere quale mosse sia la migliore per ogni stato possibile.

Per rappresentare questo algoritmo con una formula scriviamo una somma di ricompense che otteniamo dopo ogni azione, ma moltiplicheremo ogni membro futuro con γ :

$$Q(s,a)=r_0+\gamma r_1+\gamma^2 r_2+\gamma^3 r_3+\dots$$

γ è chiamato 'discount factor' ed è impostato come $0 < \gamma < 1$, e si assicura che la somma nella formula sia finita. Il valore di ciascun membro diminuisce esponenzialmente quanto sono sempre più nel futuro e diventa zero nel limite. Pertanto, γ controlla quanto la funzione Q nello stato s dipende dal futuro, e quindi può essere considerata quanto avanti vede l'agente. In genere lo impostiamo su un valore vicino, ma inferiore a uno.

Da questa possiamo ricavare la 'Bellman equation', cioè la funzione che è stata implementata in questo progetto:

$$Q(s,a)=r_0+\gamma(r_1+\gamma r_2+\gamma^2 r_3+\dots)=r_0+\gamma \max_a Q(s',a)$$

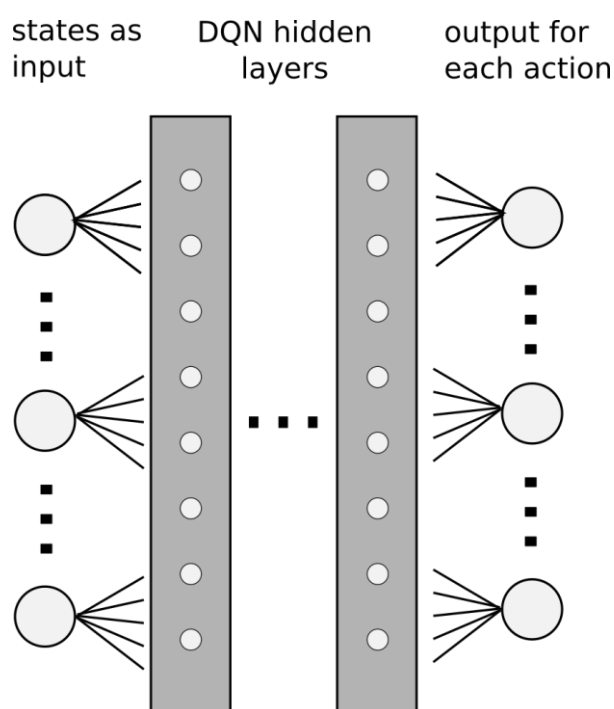
Le azioni sono scelte secondo la politica 'greedy', scegliendo l'azione che in questo momento massimizza la funzione Q .

Potremo usare questa formula ogni volta che il nostro agente sperimenta una nuova transazione e nel tempo convergerà alla funzione Q^* , cioè la funzione che sceglie la miglior mossa per ogni stato.

Tuttavia, in questo tipo di problemi, lo stato di solito consiste in svariate possibili situazioni della scacchiera e quindi il nostro spazio degli stati è molto grande.

Ovviamente non possiamo usare una tabella per memorizzare un numero quasi infinito di valori. Invece, approssimeremo la funzione Q con una rete neurale.

Questa rete prenderà uno stato come input e produrrà una stima della funzione Q per ogni azione. E se usiamo diversi livelli, il nome diviene - Deep Q-network (DQN).



L'algoritmo Q-learning necessita, per le sue caratteristiche, di un numero alto di esempi per riuscire ad imparare in maniera efficiente il suo scopo. Questo poiché risulta difficile estrarre da ogni campione tutto il valore che esso possiede.

Se immaginiamo di imparare a provare a giocare a PONG per la prima volta, come umani ci occorrerebbero pochi secondi per imparare a giocare basandosi su pochissimi campioni. Questo ci rende molto "campione efficienti". I moderni algoritmi Reinforcement learning dovrebbero vedere 100mila volte più dati, quindi sono, relativamente, campione inefficienti.

In questo caso di apprendimento non tutti i campioni sono utili allo stesso modo in quanto alcuni non fanno parte della distribuzione a cui siamo interessati. Utilizzerò alcune tecniche per filtrare questi campioni.

Nonostante ciò però il problema permane anche nel mio progetto, infatti per allenare in mio agente ci sarà la necessità di effettuare diverse decine di migliaia di partite; questo fatto è aggravato dalla complessità del gioco che risulta non banale e con un numero di stati e mosse possibili elevato.

Double DQN:

Per migliorare le prestazioni di una rete neurale Deep Q-network ho implementato una Double DQN (DDQN), che cerca di risolvere alcuni problemi delle reti neurali.

Si può notare come nell'algoritmo del Q-learning la rete neurale lavora come un tutt'uno, infatti ogni aggiornamento della funzione Q influenza anche l'intera rete.

I punti di $Q(s, a)$ e $Q(s', a)$ sono molto simili tra loro, perché ogni transazione descrive una mossa che porta da s a s' . Questo porta a un problema, che con ogni aggiornamento è probabile che la rete venga modificata in maniera pesante. Come si può immaginare, questo può portare a instabilità, oscillazioni o divergenze.

Per ovviare a questo problema, varie ricerche hanno proposto di utilizzare una rete 'destinazione' separata. Questa rete è una semplice copia della rete precedente, ma congelata nel tempo. Fornisce valori Q^{\sim} stabili e consente all'algoritmo di convergere verso il target specificato:

$$Q(s,a) \rightarrow r + \gamma \max_a Q^{\sim}(s',a)$$

Dopo diversi passaggi, la rete di destinazione viene aggiornata, semplicemente copiando i pesi dalla rete corrente. Per essere efficace, l'intervallo tra gli aggiornamenti deve essere abbastanza grande da lasciare abbastanza tempo per la convergenza della rete originale.

Uno svantaggio è che rallenta sostanzialmente il processo di apprendimento.

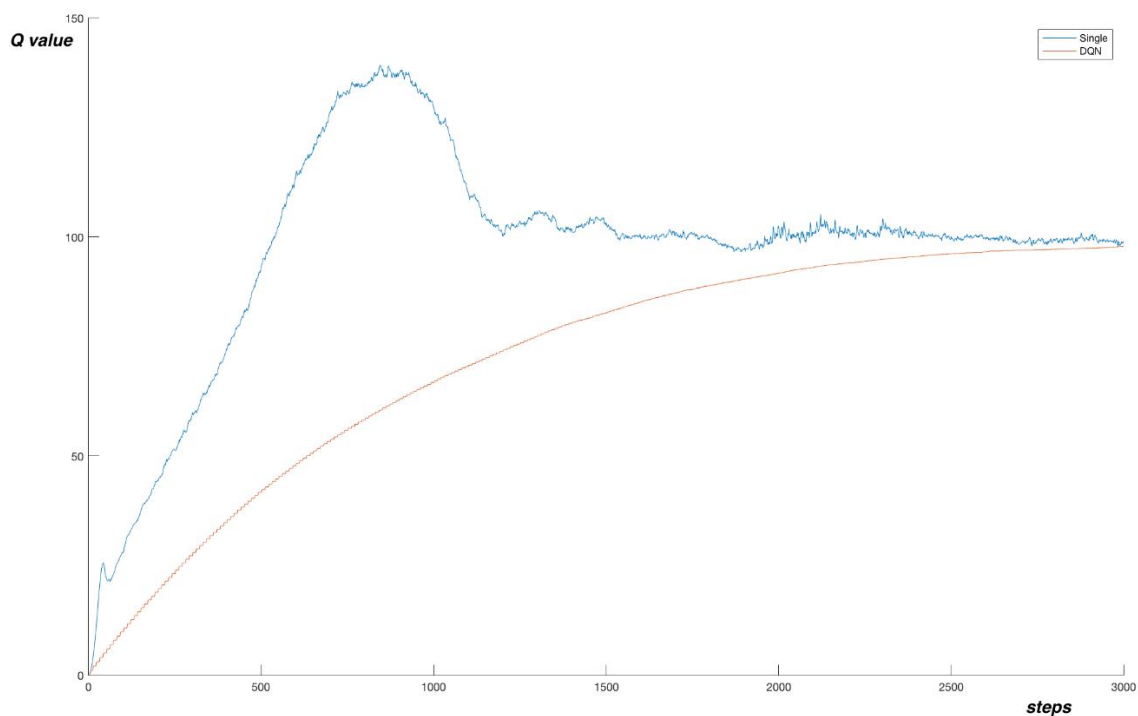
Qualsiasi modifica alla funzione Q viene propagata solo dopo l'aggiornamento della

rete di destinazione. Gli intervalli tra gli aggiornamenti sono in genere in ordine di migliaia di passaggi, quindi questo può davvero rallentare le cose.

Ora presento un grafico ottenuto da un forum ^[2] dove vengono comparate una rete Q-learning semplice (blu) e una rete Double Q-network (arancione).

La versione con la rete di destinazione mira senza problemi al vero valore, mentre la semplice rete Q-learning mostra alcune oscillazioni e difficoltà.

Pur sacrificando la velocità di apprendimento, questa stabilità aggiuntiva consente all'algorithmo di apprendere il comportamento corretto in ambienti molto complicati, come giocare a giochi da tavolo o ai giochi Atari che ricevono solo input visivi.



Un altro problema che il Double DQN tenta di risolvere è che nell'algorithmo DQN l'agente tende a sopravvalutare il valore della funzione Q, a causa del massimo nella formula utilizzata per impostare gli obiettivi.

Per dimostrare questo problema, si può immaginare una situazione come la seguente. Per uno stato particolare esiste una serie di azioni, tutte con lo stesso valore Q reale. Ma la stima è intrinsecamente rumorosa e differisce dal valore reale. A causa del massimo nella formula, viene selezionata l'azione con l'errore positivo più elevato e questo valore viene successivamente propagato ulteriormente ad altri stati. Ciò porta a una distorsione positiva, cioè una sopravvalutazione del valore. Questo ha un grave impatto sulla stabilità del nostro algorithmo di apprendimento. Una soluzione a questo problema è stata proposta da Hado van Hasselt nel 2010 ed è stata chiamata Double Learning.

In questo nuovo algorithmo, due funzioni Q, Q1 e Q2, vengono apprese in modo indipendente. Una funzione viene quindi utilizzata per determinare l'azione di massimizzazione e l'altra in secondo luogo per stimarne il valore.

Nel nostro caso possiamo sfruttare il fatto che abbiamo già due reti diverse dandoci due stime diverse Q e Q^\sim (rete destinazione). Sebbene non sia realmente indipendente, ci consente di cambiare l'algoritmo in un modo davvero semplice. La formula di destinazione originale diventa:

$$Q(s,a) \rightarrow r + \gamma Q^\sim(s', \operatorname{argmax}_a Q(s',a))$$

Il documento Deep Reinforcement Learning con Double Q-learning ^[3] riporta che sebbene il Double DQN non migliora sempre le prestazioni, apporta vantaggi sostanziali alla stabilità dell'apprendimento. Questa stabilità migliorata si traduce direttamente nella capacità di apprendere compiti molto complicati. Per questo motivo questa implementazione mi è sembrata la più adatta per il mio scopo.

Prioritized Experience Replay:

Durante ogni fase di simulazione, l'agente esegue un'azione a in uno stato s , riceve una ricompensa immediata r e arriva a un nuovo stato s' . Si può notare che questo schema che si ripete può essere scritto come (s, a, r, s') .

Una delle possibili tecniche è l'apprendimento online, cioè l'utilizzo delle singole situazioni per imparare immediatamente da esse. Però con questo tipo di apprendimento risultano esserci alcuni problemi: i campioni arrivano nell'ordine in cui sono stati trovati e come tali sono altamente correlati. Per questo motivo, molto probabilmente la nostra rete avrà problemi di 'overfitting' e non riuscirà a generalizzare correttamente; il secondo problema è che non stiamo usando la nostra esperienza in modo efficace, in realtà buttiamo via ogni campione immediatamente dopo averlo usato.

Per questi motivi ho scelto di utilizzare un Prioritized experience replay dove ad ogni mossa alleno il modello su una minibatch estratta dalla memoria permettendo di avere dati potenzialmente non correlati e una maggior varietà. In particolare, la PER sfrutta il fatto che si possa apprendere maggiore conoscenza da alcune transazioni rispetto che da altre e tenta di sfruttare questo fatto modificando la distribuzione di campionamento.

L'idea principale è che preferiamo le transazioni che non si adattano bene alla nostra attuale stima della funzione Q , perché queste sono le transazioni da cui possiamo imparare di più. Questo si basa su una semplice intuizione dal nostro mondo reale: se incontriamo una situazione che differisce veramente dalle nostre aspettative, ci pensiamo più volte e cambiamo il nostro modello fino a quando non si adatta. Per la nostra rete possiamo definire l'errore di un campione $S = (s, a, r, s')$ come distanza tra il valore di $Q(s,a)$ e il suo obiettivo $T(S)$, dove T per la nostra rete è:

$$T(S)=r+\gamma Q_{\sim}(s',\operatorname{argmax}_a Q(s',a))$$

Conserveremo questo errore nella memoria dell'agente insieme a ogni campione e lo aggiorneremo ad ogni fase di apprendimento.

L'approccio che ho scelto è chiamato 'prioritizzazione proporzionale' dove l'errore è trasformato in una priorità seguendo questa formula:

$$p=(\text{error}+\epsilon)^\alpha$$

Epsilon ϵ è una piccola costante positiva che assicura che nessuna transazione abbia priorità zero. Alpha, $0\leq\alpha\leq1$, controlla la differenza tra errore alto e basso, cioè determina la quantità di priorità utilizzata. Con $\alpha = 0$ otterremmo il caso uniforme. Successivamente la priorità calcolata viene trasformata nella probabilità di essere scelto per l'apprendimento. Un campione i ha probabilità di essere scelto durante il processo di Experience Replay determinato da una formula:

$$P_i=p_i\sum_k p_k$$

L'algoritmo è semplice: durante ogni fase di apprendimento otterremo una serie di campioni con questa distribuzione di probabilità e formeremo la nostra rete su di essi. Abbiamo solo bisogno di un modo efficace per memorizzare queste priorità e campionarle da esse.

Per realizzarlo ho scelto di utilizzare due classi Memory e SumTree ^[4]. La classe Memory si occupa di funzioni di alto livello ed è la classe che possiede il mio agente e con la quale si interfaccia; per esempio permette di aggiungere o aggiornare i dati, oppure di creare una minibatch di transazioni con le quali fare l'apprendimento.

La classe SumTree si occupa dell'implementazione vera e propria dell'albero all'interno del quale verranno memorizzati i nostri stati, con le rispettive azioni e ricompense ottenute, e che sta alla base del metodo appena descritto per applicare la tecnica del Prioritized experience replay.

Rete Neurale:

La rete Neurale che ho implementato è realizzata da diversi livelli sequenziali e densi utilizzando le primitive del framework Keras. Come funzione di attivazione dei vari livelli ho utilizzato la funzione 'Relu', che restituisce il valore massimo fra 0 e il valore passatogli in ingresso. Per l'ultimo livello della rete che restituisce i valori ho deciso

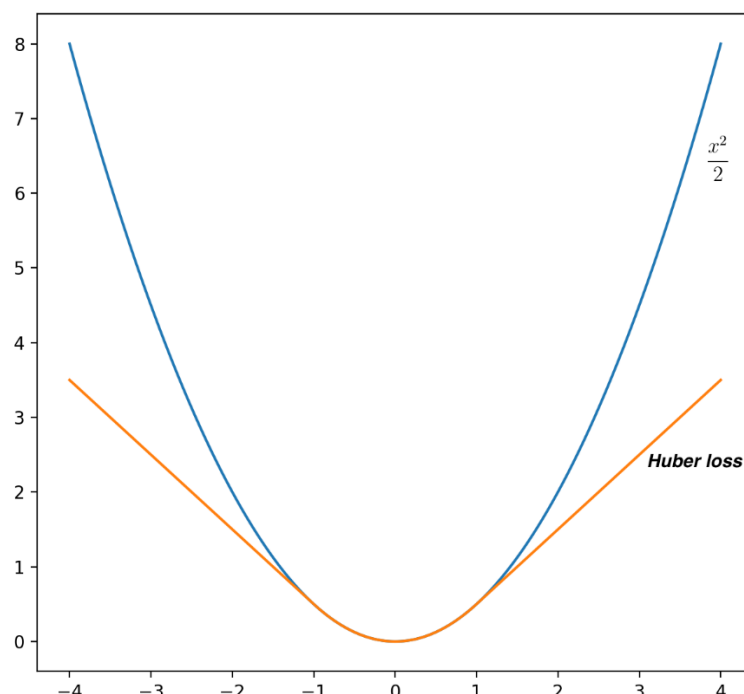
di usare la funzione di attivazione 'Softmax', che prende in ingresso un vettore di numeri reali e restituisce un vettore costituito dalle probabilità proporzionali per ogni elemento del vettore in ingresso.

Come ingresso alla rete verrà passato lo stato della scacchiera; quindi verrà passato un vettore di 64 elementi, rappresentanti le 64 caselle della scacchiera, dove uno 0 rappresenta una casella vuota, un 1 una casella occupata dal nero e un 2 una casella occupata dal bianco. Come già descritto, in uscita la rete restituirà un vettore di altri 64 elementi, dove l'elemento con il valore più alto descrive la posizione della prossima casella da riempire con una nostra pedina.

Si è scelto di passare alla rete in ingresso tutta la scacchiera, quindi senza dare nessun aiuto alla nostra rete che dovrà capire tramite le ricompense anche quali mosse sono valide oppure no. Un'altra possibilità era di dare in ingresso solamente le posizioni delle mosse valide per quello stato, ma non è stata scelta questa strada. Principalmente perché mi sembrava più confusionaria, in particolar modo dato che il numero di posizioni valide varia notevolmente in base al momento della partita e alla composizione delle pedine sulla scacchiera.

Come 'loss function' della rete è stata scelta la Huber Loss Function, una funzione di perdita molto utilizzata in ambiti simili a quello di questo progetto, che è meno sensibile ai valori anomali nei dati rispetto alla classica MSE.

Di seguito presento un grafico del confronto preso dalla fonte ^[2]:



Come ricompense per la rete ho deciso di mantenere il range fra -1 e 1, per rendere piccolo anche il range della funzione obiettivo e di conseguenza anche il gradiente che si calcola durante l'apprendimento.

Viene assegnata una ricompensa negativa se si è stati sconfitti oppure se è stata scelta una mossa invalida, mentre una positiva se si è vinta la partita oppure se è stata scelta una mossa valida.

Fase di apprendimento:

Prima di ogni serie di partite di apprendimento ho deciso di effettuare una serie di sfide solamente utilizzando i giocatori casuali descritti in precedenza; questi giocatori si scontrano e vengono sfruttati i match per riempire la memoria PER con dati reali che permettono di velocizzare la fase di apprendimento della rete. Infatti, la memoria risulterà già inizializzata e di conseguenza anche le prime azioni di apprendimento risultano essere significative.

Si è deciso che rispetto al nostro agente, anche le azioni del giocatore casuale faranno parte dell'ambiente e quindi lo stato s' che verrà passato alla rete rappresenterà la scacchiera dopo la mossa effettuata dall'avversario.

Successivamente a questa fase preliminare, verrà inizializzato il nostro agente che si troverà quindi ad effettuare una serie di partite in sequenza sempre contro il giocatore casuale da noi creato. Per ogni mossa del nostro agente memorizzeremo i dati di questa transazione e avvieremo un passo dell'apprendimento per poi continuare la partita oppure passare ad una successiva. Il passo di apprendimento riguarda l'utilizzo della PER per ricevere una minibatch sulla quale effettuare le operazioni descritte in precedenza caratteristiche di una rete Double DQN.

Per la gestione della partita dell'agente è stata implementata una politica epsilon-greedy, cioè la scelta di una mossa in base ad una probabilità. Si seleziona la mossa predetta dalla rete con probabilità $(1-\epsilon)$, e una casuale con probabilità ϵ . Inizialmente epsilon viene impostato dal nostro agente come 1 e cala man mano che l'episodio avanza. In questa maniera si offre un grado maggiore di esplorazione all'agente per individuare mosse più interessanti per cercare di velocizzare l'apprendimento e renderlo più completo esplorando diverse soluzioni. In questo caso le mosse casuali del nostro agente saranno realmente casuali fra qualsiasi casella della scacchiera, al contrario di quelle del giocatore avversario che sceglie solo fra quelle valide. Questo è stato scelto per permettere al nostro agente di esplorare anche le mosse invalide, che altrimenti sarebbero difficilmente esaminate, e quindi riuscire a crearsi una migliore idea della scacchiera e di quali siano le mosse valide per tutti i possibili stati.

Conclusioni:

Con la struttura appena descritta sono riuscito a creare un agente che, definita una rete neurale precisa, delle ricompense e una serie di parametri, riesce ad imparare a giocare ad Othello senza avere nessun tipo di spiegazione delle regole del gioco.

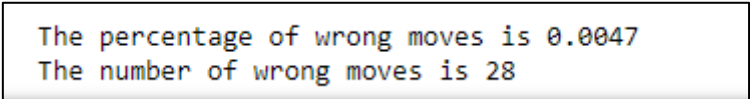
Nel nostro agente non è presente nessun tipo di euristica, inoltre non viene spiegato in nessun modo, a livello di codice, quali mosse possano essere valide e quali no;

l'unico modo per orientare l'apprendimento è tramite le ricompense date alla rete.

Per permettere alla rete di affrontare una complessità minore è stata creata una nuova classe chiamata 'SmallBoard', cioè una piattaforma di gioco uguale alla originale ma dove la scacchiera risulta essere di 4x4 caselle. Si è deciso di effettuare questa semplificazione poiché le risorse hardware in mio possesso non sono estremamente performanti e in questo modo ho potuto testare il mio agente in un ambiente agevolato permettendo una migliore visione dei risultati, che altrimenti rimanevano nascosti dalla complessità del gioco.

Grazie a questa manipolazione sono riuscito, attraverso le dovute modifiche, a creare un agente che potesse apprendere più velocemente e al meglio il gioco aiutato dal minor numero di stati e possibilità di mosse.

L'agente risulta essere performante successivamente ad un episodio di apprendimento di 7000 partite, la cui durata con il mio hardware è di circa un'ora. Al completamento di questa fase l'agente risulta essere in grado di sostenere partite del gioco Othello e di scegliere in maniera autonoma le mosse da eseguire incappando in mosse non valide con probabilità molto basse; durante una serie di 1000 partite risultano non valide circa 25 mosse sulle 6000 effettuate.



```
The percentage of wrong moves is 0.0047
The number of wrong moves is 28
```

Inoltre, sempre all'interno di questi episodi contro i giocatori casuali del nostro ambiente, l'agente risulta vincitore con una probabilità di circa il 50% (Grafico 2).

Ora vengono mostrati i grafici del giocatore che è risultato meglio performante successivamente ai vari episodi da me eseguiti.

Il primo grafico presenta un episodio di apprendimento durante il quale viene salvato il numero di mosse non valide effettuate; per ogni partita viene salvato un '1' se è stata commessa una mossa non valida, mentre uno '0' se la partita si è conclusa normalmente senza difficoltà. Come già descritto, l'agente effettua una fase di esplorazione iniziale dove vengono scelte quasi sempre mosse casuali e questo comporta la massa iniziale di mosse non valide presente circa fino alle partite numero 3000.

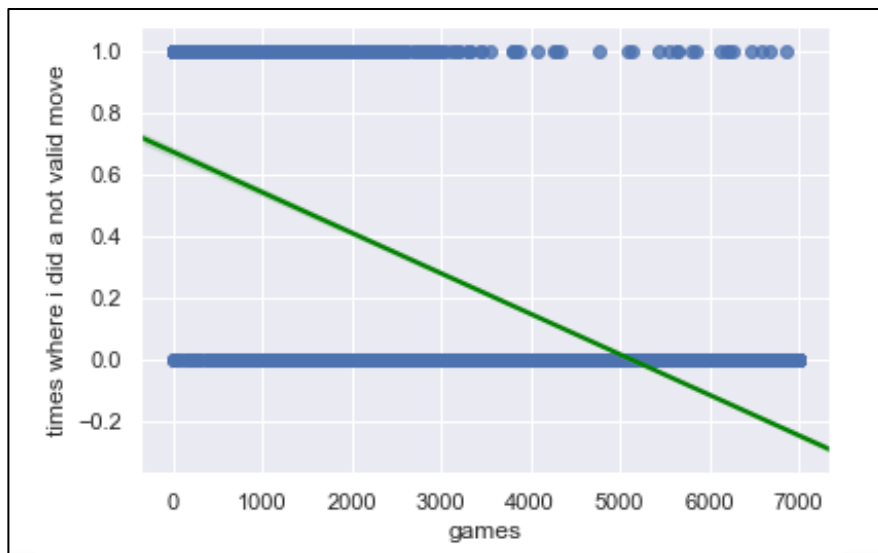


Grafico1

Il secondo grafico presenta l'agente che, dopo l'episodio di apprendimento, effettua una serie di 1000 partite contro il mio giocatore casuale e durante le quali viene salvato il numero di vittorie dell'agente; per ogni partita viene salvato un '1' se è stata vinta mentre uno '0' se si è usciti sconfitti, durante queste partite quando il nostro giocatore sceglie una mossa non valida viene considerata una sconfitta.

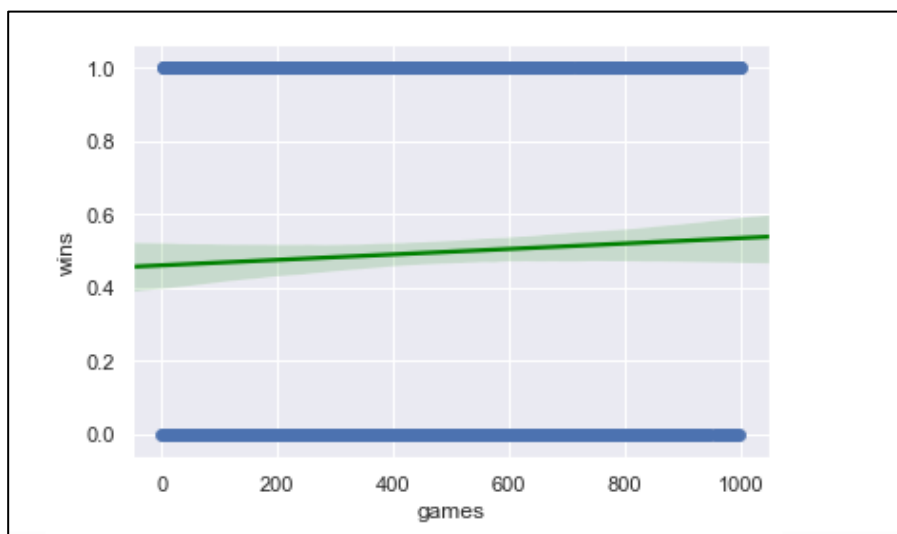


Grafico 2

Per comparazione, l'agente che gioca nella scacchiera originale effettuando una sessione di apprendimento equivalente a quella appena descritta risulta avere in ogni caso un buon miglioramento ma continua comunque a effettuare diverse mosse non valide. Il grafico seguente presenta un episodio di apprendimento durante il quale viene salvato il numero di mosse valide predette dall'agente prima di un errore. Come per l'agente semplificato la massa iniziale di mosse predette molto basse è dovuta alla fase di esplorazione; successivamente invece le mosse casuali quasi si azzerano e si può notare un miglioramento graduale, ma molto lento, che continua fino alla fine dell'episodio.

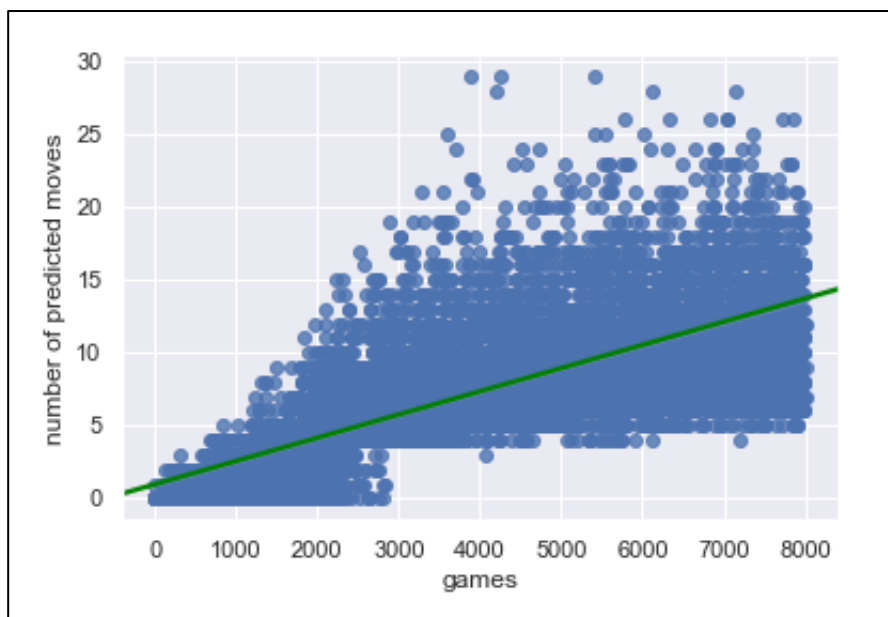


Grafico 3

Un altro confronto interessante è sullo sviluppo della funzione Q da parte del nostro agente. Di seguito presento una immagine illustrativa che mette a confronto la funzione Q su una medesima configurazione della scacchiera tra l'agente performante, su cui è stato fatto un episodio di apprendimento della durata di 7000 partite, e un altro agente che ha effettuato un episodio di solamente 100 partite. Si può notare come nella seconda rete la veloce fase di apprendimento non permette alla rete di distinguere le mosse valide da quelle non valide, infatti vediamo che i valori della funzione Q sono molto simili per tutte le caselle e quella a valore maggiore non è una mossa valida. Al contrario la prima rete risulta avere una varietà maggiore: si può notare un numero bassissimo nelle aree centrali, cioè le quattro caselle che risultano sempre occupate anche appena iniziata la partita; inoltre si possono notare varie caselle con numeri intermedi, più bassi se caselle facilmente eliminabili o più alti se mosse valide o simili, però si distacca molto dalle altre la mossa scelta, che risulta per la rete la mossa nettamente migliore.

1	•	•	•	•
2	•	•	•	•
3	•	•	•	•
4	•	•	•	•
	A	B	C	D
Q-function of trained agent:				
7.788171e-12	5.9560186e-16	3.6487373e-09	3.5780331e-10	
3.3457832e-07	2.365053e-21	3.6767714e-21	2.3967648e-09	
5.2521425e-11	1.5561957e-21	2.2788872e-21	5.2619168e-08	
3.0647536e-06	0.99999034	5.3167e-07	5.677239e-06	
Q-function of short trained agent:				
0.039662585	0.064819194	0.016968185	0.076259896	
0.21008962	0.009687764	0.0060026348	0.024160553	
0.01781183	0.009611703	0.009583539	0.15350941	
0.16079202	0.029436512	0.1444915	0.027113046	

Questi dati mettono le basi per la creazione di un agente in grado di giocatore anche ad Othello con scacchiera 8x8 utilizzando hardware più potenti ed episodi più longevi. Inoltre, attraverso sviluppi futuri si potrebbe perfezionare la rete e velocizzarne il suo apprendimento per renderla più performante.

Fonti:

[1] - <http://realerthinks.com/creating-othello-tutor-step-1-making-playable-game>

[2] - <https://jaromiru.com/2016/10/21/lets-make-a-dqn-full-dqn/>

[3] - Hado van Hasselt, Arthur Guez, David Silver - *Deep Reinforcement Learning with Double Q-learning*, 2016

[4] - <https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/>