SAPIENZA
UNIVERSITÀ DI ROMA

Reinforcement Learning nell'AI: Implementazione dell'Algoritmo AlphaZero

Facoltà di Ingegneria dell'informazione, informatica e statistica
Dipartimento di Ingegneria informatica, automatica e gestionale
Corso di laurea in Ingegneria Informatica e Automatica

Pietro Spadaccino
Matricola 1706250

Relatore
Fiora Pirri

A.A. 2017-2018

To live is to risk it all, otherwise you're just an inert chunk of randomly assembled molecules drifting wherever the universe blows you

- Rick Sanchez

Indice

Abstract

L'applicazione dell'intelligenza artificiale ai giochi è sempre stato un campo di grande interesse. La maggior parte dei programmi viene solitamente scritta specificamente per il proprio gioco, cercando di migliorare il più possibile le proprie prestazioni oppure di ottimizzare la fase di allenamento, se necessaria. AlphaGo Zero e il successivo algoritmo generale AlphaZero, stravolgono questo aspetto, riuscendo ad ottenere prestazioni anche migliori di quelle umane nel gioco. Lo scopo di questo progetto è di illustrare le idee alla base di AlphaZero, presentare un'implementazione e allenare una policy di gioco con tale algoritmo.

1. Introduzione

AlphaGo è stato il primo programma che ha battuto un campione mondiale nel gioco del Go. Creato da Deepmind, è stato allenato in maniera supervisionata usando due singole *deep neural networks* per valutare e selezionare le mosse, cercando di imitare le mosse di giocatori esperti e perfezionato tramite reinforcement learning. Successivamente è stato presentato AlphaGo Zero, un algoritmo basato esclusivamente sul reinforcement learning che ha battuto 100-0 il precedente AlphaGo. AlphaGo Zero ha raggiunto livelli elevatissimi pur non avendo mai “visto” partite giocate né dai campioni mondiali né tanto meno dal precedente AlphaGo, ma solo giocando partite contro sé stesso. Si basa su una sola rete neurale, che seleziona le mosse durante la partita e che predice il vincitore della stessa, guidata dal il Monte Carlo Tree Search (MCTS), e viene allenata sulle partite svolte contro se stessa. Deepmind successivamente presentato AlphaZero, l’algoritmo di AlphaGo Zero, attraverso il quale sono state allenate policy che hanno raggiunto livelli elevatissimi nel gioco degli Scacchi e del Shogi. La logica del nuovo AlphaZero

è infatti invariante rispetto al gioco o al dominio di applicazione, ed è dipendente solo dalle regole dello stesso.

1.1 Intelligenza artificiale nei giochi

L'applicazione dell'intelligenza artificiale ai giochi, nello specifico giochi da tavolo, ha sempre avuto grande ricerca a seguito. Basti pensare a figure come Turing, Shannon o von Neumann che svilupparono algoritmi e hardware per analizzare il gioco degli Scacchi. I giochi da tavolo sono stati, e sono tutt'ora, una sfida per generazioni di ricercatori. Col passare del tempo nacquero diversi programmi e algoritmi che raggiunsero livelli altissimi, come il motore di gioco *Stockfish* per Scacchi. Questi sistemi sono estremamente ottimizzati per il proprio gioco, riuscendo ad estrapolare dallo stato della partita *features* difficili da osservare anche per i giocatori più esperti. Tuttavia questa ottimizzazione rende gli algoritmi in questione non generalizzabili per altri contesti. Un traguardo dell'intelligenza artificiale fu raggiunto nel 1997 quando *Deep Blue* sconfisse per la prima volta un campione mondiale. Anche quest'ultimo fu altamente adattato al gioco degli Scacchi, utilizzando euristiche proprie del dominio.

Game	White	Black	Win	Draw	Loss
Chess	*AlphaZero*	*Stockfish*	25	25	0
	Stockfish	*AlphaZero*	3	47	0
Shogi	*AlphaZero*	*Elmo*	43	2	5
	Elmo	*AlphaZero*	47	0	3
Go	*AlphaZero*	*AG0 3-day*	31	–	19
	AG0 3-day	*AlphaZero*	29	–	21

Figura 1: Risultati di AlphaZero applicato a giochi diversi. AG0 3-day indica AlphaGo Zero con tre giorni di training. Immagine tratta da referenza (1)

Shogi è un altro gioco da tavolo che, come Scacchi, ha avuto molta ricerca a seguito. E' un gioco computazionalmente più complesso degli Scacchi, utilizzando una scacchiera più grande e avendo regole difficili da "insegnare" ad agenti software, come la possibilità che ha ogni pedina eliminata di essere rimessa sul campo di gioco in qualsiasi posizione.

Rispetto a Scacchi e Shogi, il Go si combina bene con l'architettura delle reti neurali, essendo invariante a traslazioni dello stato di gioco, rispecchiando quindi il funzionamento dei layer convoluzionali. Ciò nonostante tramite l'algoritmo AlphaZero sono state allenate policy per Scacchi e Shogi che hanno raggiunto livelli estremamente alti, riuscendo a competere con i programmi più forti del proprio gioco, vincendo *28 / 100* partite (pareggiando le restanti *72*) contro *Stockfish* e *90 / 100* partite contro *Elmo*, uno dei motori di gioco più forti di Shogi.

L'algoritmo AlphaZero è di fatto usabile in qualsiasi gioco o contesto totalmente visibile, non avendo logiche che dipendono da nessun dominio particolare.

1.2 Metodi comunemente applicati

L'intelligenza artificiale applicata ai giochi spesso utilizza metodi simili tra di loro, come l'uso di funzioni di valutazione dello stato di gioco oppure ricerche dell'albero. Una funzione di valutazione prende come argomento lo stato di una partita e restituisce un valore, spesso uno scalare, che rappresenta la "bontà" dello stato. Un valore alto indica un buon posizionamento del giocatore rispetto all'avversario e quindi alte possibilità di vittoria. Le funzioni di valutazione sono spesso utilizzate insieme a ricerche dell'albero di gioco per valutare il nodo (lo stato del gioco) corrente. Si noti che a volte possono non avere costo computazionale costante, risultando onerose se si esegue la valutazione su molti nodi.

Uno degli algoritmi base per effettuare una ricerca nell'albero di gioco è il *Minimax*, chiamato così perché cerca di *mini*mizzare la perdita nel caso di

max loss. Esso consiste nel massimizzare la vincita dell'avversario e successivamente scegliere la mossa che minimizza la perdita per sé stesso in base all'analisi del caso peggiore, cioè il caso in cui l'avversario giochi la mossa migliore, dove vincita e perdita sono valori calcolati con funzioni di valutazione. Intuitivamente, per scegliere una mossa viene simulato lo stato del gioco per tutte le possibili mosse, e in questi stati viene simulata la mossa dell'avversario in modo che massimizzi il suo profitto, cioè giochi al meglio. A questo punto si seleziona la mossa da eseguire (nella partita originale) che minimizzi la vincita dell'avversario.

Tecnicamente, il *Minimax* espande tutti gli stati del gioco eseguendo una DFS o BFS, riuscendo a prevedere l'andamento

del gioco.

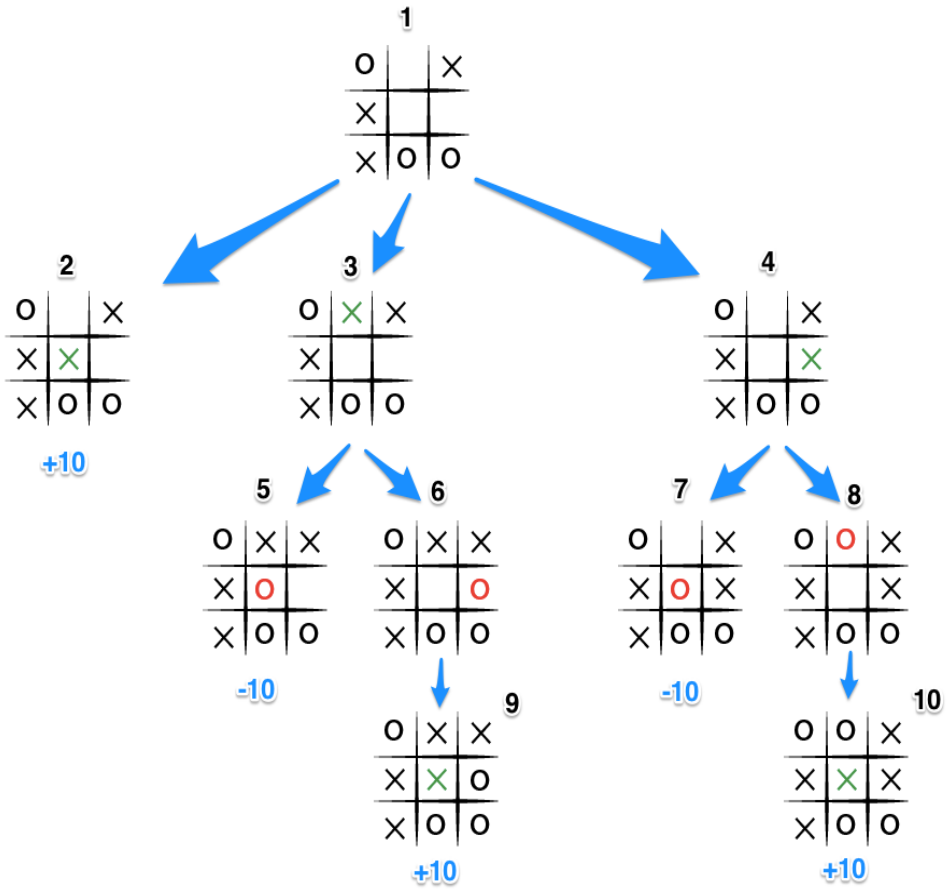

Figura 2: Simulazione del Minimax su una partita di tris. Immagine tratta da neverstopbuilding.com

Il maggior punto debole del *Minimax* è la quantità esponenziale di mosse da valutare durante la visita dell'albero, tant'è vero che la DFS o BFS viene limitata ad un certo livello di esplorazione. Ad esempio, prendiamo una partita

di Scacchi dove un giocatore può muovere *16* pezzi totali: l'albero quindi avrà un *branching factor* pari a *16*. Per arrivare a profondità *8*, che equivale a prevedere solamente *4* mosse future dell'avversario, sono necessarie più di $4x10^9$ valutazioni. Il *Minimax* e in generale le ricerche dell'albero complete possono essere migliorate scartando alcuni rami quando ci si accorge della perdita eccessivamente negativa, ma il costo esponenziale intrinseco li rende proibitivi se non per giochi poco complessi.

2. Reinforcement Learning

Il *Reinforcement Learning* (RL) è una branca del *machine learning* che si occupa di far apprendere ad agenti software come eseguire delle azioni in un ambiente (*environment*) in modo da massimizzare una certa ricompensa (*reward*).
E' diverso dal *supervised learning* classico poiché non si hanno a disposizione esempi di input/output su cui allenare la rete. Nel RL invece è la policy stessa che deve trovare il modo migliore di eseguire le azioni, cercando di bilanciare tra *exploration* e *exploitation*, ovvero l'esplorazione di nuovi stati non ancora visitati e lo sfruttamento degli stati conosciuti (3).

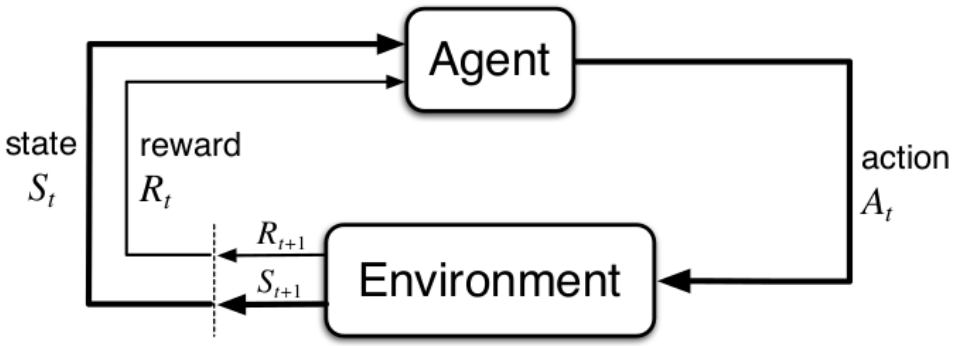

Figura 3: Struttura base di un algoritmo di Reinforcement Learning: un agent esegue un'azione su un environment, ricevendo indietro una ricompensa dell'azione e il nuovo stato attuale. Immagine tratta da towardsdatascience.com

Il *Q-learning* è una delle tecniche più utilizzate del Reinforcement Learning. Esso comprende un agente, un insieme di stati *S* e un insieme di azioni *A* per ogni stato. Eseguendo un'azione *a* l'agente passa da uno stato *s* a uno stato *s'*, e gli viene assegnata una ricompensa. L'obiettivo dell'agente è quello di massimizzare la sua reward futura, sommando la ricompensa ottenuta dal raggiungimento dello stato *s'* e la massima ricompensa ottenibile dagli stati futuri. Il Q-learning tiene traccia delle reward per ogni stato attraverso una

tabella di dimensioni *(stati x azioni)* inizialmente posta a zero, e aggiornando i valori delle ricompense iterativamente secondo la formula:

$$Q(s_t, a) = Q(s_t, a)(1-\alpha) + \alpha(r_t + \gamma * maxQ(s_{t+1}, a))$$

dove r_t è la reward per lo stato corrente s_t, α rappresenta il learning rate e γ è il fattore di sconto, che controlla il valore delle ricompense future.

Tuttavia, in ambienti con tanti stati possibili, utilizzare una tabella delle ricompense potrebbe essere inefficiente o richiedere troppa memoria. Basti pensare al caso di Scacchi dove sono possibili un numero di posizioni di gioco nell'ordine di 10^{45}. Da qui nasce una variante del Q-learning, il *deep* Q-learning, che utilizza una rete neurale per "simulare" la tabella delle rewards. Per far apprendere i valori si utilizza una tecnica chiamata *experience replay*, allenando la rete su partite svolte da sé stessa. Questa tecnica viene utilizzata anche in AlphaZero.

2.1 Reinforcement Learning in AlphaZero

AlphaZero allena la policy di gioco sui dati delle partite giocate contro sé stessa attraverso un nuovo algoritmo di reinforcement learning. Esso è diviso in due fasi principali: la fase di *selfplay* e la fase di *training*. Durante la prima la policy esegue delle partite contro sé stessa, utilizzando il Monte Carlo Tree Search guidato da una singola rete neurale per selezionare ogni mossa, mentre durante la fase di training la rete viene allenata sui dati di gioco generati. Una volta terminate entrambe le fasi si inizia una nuova iterazione usando la nuova rete allenata.

La rete neurale di AlphaZero f_θ riceve in input lo stato del gioco S, e restituisce in output due valori: $(P, v) = f_\theta(S)$ dove P è un vettore che indica la probabilità che ha ogni mossa di essere scelta e v è uno scalare che stima la "bontà" dello stato S.

In ogni stato del gioco s_t, per scegliere la mossa da eseguire, viene eseguita una ricerca dell'albero attraverso il Monte Carlo Tree Search, guidata dalla rete f_θ. Il vettore di probabilità ritornato in output dal MCTS $\Pi_t = MCTS(s_t, P)$ indica mosse molto più "forti" rispetto a P ritornato dalla rete. Il MCTS può quindi essere visto come un potente mezzo per migliorare le predizioni della rete. Si noti che data la dipendenza di Π da P si ha un miglioramento non eccessivo rispetto a P: in questo modo la rete riesce ad imparare passo passo, come se si stesse allenando su partite svolte da giocatori con un livello via via crescente.

2.1.1 Selfplay

Durante la fase di selfplay si giocano delle partite dove i due avversari seguono la stessa rete, utilizzando le probabilità del MCTS per eseguire ogni mossa. Le partite vengono memorizzate e la rete viene allenata in modo supervisionato tramite *backpropagation* (vedi sezione Backpropagation in Metodi), in modo da minimizzare l'errore tra le sue probabilità P e le probabilità del MCTS Π. Più precisamente, dato che la rete ha due output, viene minimizzato l'errore tra (P, v) e (Π, z), indicando con z il vincitore della partita. Il selfplay e il training vero e proprio della rete si susseguono in maniera iterativa: la nuova rete allenata viene confrontata contro la rete precedente e se vince la maggior parte delle partite allora viene accettata e viene utilizzata per generare nuovi dati di training (nella fase di selfplay) durante l'iterazione successiva, altrimenti viene scartata. Si noti che si può anche omettere il passo della valutazione ed accettare a priori la nuova rete.

Il MCTS che viene utilizzato è composto da nodi, che rappresentano gli stati della partita, e da archi (s, a) che collegano due nodi, rappresentanti l'azione a attuata allo stato s. Ogni arco conserva:

- la sua probabilità a priori $P(s, a)$ di essere selezionato, ovvero la probabilità di scegliere l'azione a dallo stato s *secondo la rete neurale*;
- un contatore di visite $N(s, a)$, ovvero quante volte l'arco (s, a) è stato visitato dal MCTS durante le simulazioni;

- il q-value $Q(s,a)$, quanto è "buono" scegliere l'arco (s,a) per il giocatore corrente.

Ogni simulazione del MCTS inizia dalla radice e seleziona via via le azioni che massimizzano $Q(s,a)+U(s,a)$, dove $U(s,a) \propto \frac{P(s,a)}{1+N(s,a)}$. Ciò significa che $U(s,a)$ è proporzionale alla probabilità a priori di una certa azione ed è inversamente proporzionale al numero di volte che questa azione è stata visita. Se un'azione (s,a_i) è stata visitata poche volte rispetto alle altre azioni $(s,a_{j\neq i})$ allora $U(s,a_i)$ sarà più alto e l'azione sarà incentivata ad essere selezionata durante la ricerca.

Una simulazione del MCTS procede selezionando le mosse in questo modo fino a raggiungere uno stato foglia (o *leaf*) s_L : esso è una posizione ancora non visitata dal MCTS, ovvero uno stato di cui non si conoscono ancora le probabilità a priori delle sue azioni (s_L,a_i) . Lo stato allora viene valutato dalla rete neurale una sola volta $(P(s_L,a_i),v(s_L)) = f_\theta(s_L)$. Le probabilità a priori di ogni mossa a_i presenti nel vettore $P(s_L,a_i)$ vengono salvate ognuna nel suo arco (s_L,a_i) . Dato che s_L è stato ora visitato dal MCTS, non è più uno stato foglia e la fase di selezione delle mosse è terminata.

Inizia la fase di backpropagation del MCTS, nella quale ogni arco (s,a) selezionato durante la simulazione incrementa il proprio contatore di visite $N(s,a)$ di uno e aggiorna il proprio *q-value* alla media aritmetica dei *q-values* degli stati raggiunti successivamente nella ricerca dell'albero (vedi sezione Monte Carlo Tree Search in Metodi), segnando la fine di una simulazione. Si noti che nel MCTS si usano fare non una sola ma molteplici simulazioni, nell'ordine di centinaia, per scegliere una singola mossa.

Una volta terminate tutte le simulazioni, la ricerca dell'albero restituisce un vettore Π contenente le probabilità di ogni mossa a_i . Vale la relazione $\Pi_i \propto N(s,a_i)$: la probabilità di scegliere una mossa è proporzionale al numero di volte in cui è stata visitata durante le simulazioni.

2.1.2 Fase di apprendimento

Il MCTS può essere visto quindi come un algoritmo tramite il quale una policy può effettuare partite contro sé stessa, e, sulla base dei risultati, essere allenata.

Durante la fase di selfplay, all'iterazione *it,* per ogni turno della partita viene eseguito il MCTS guidato dalla rete con pesi θ_{it} . Quando una partita termina, sia per la vittoria di uno dei giocatori sia per parità, le viene attribuito un valore di reward $r \in \{-1,0,1\}$. Ogni step t della partita viene memorizzata come una tupla (s_t, Π_t, z_t) , dove $z_t = \pm r$ e vale $z_t = -z_{t+1} \forall t$. Il valore z_t è la reward del gioco a seconda della prospettiva del giocatore corrente dello step t, ovvero il vincitore del gioco a seconda del giocatore corrente. Terminate tutte le partite del selfplay si avrà un vettore di tuple:

$$[(s_0, \Pi_0, z_0),\ (s_1, \Pi_1, z_1),\ ...,\ (s_t, \Pi_t, z_t)]$$

e su questi dati verrà allenata la rete. Nello specifico la rete viene adattata per minimizzare l'errore tra il valore $v(s_t)$ predetto e la reward z_t , e per massimizzare la similarità delle probabilità delle mosse $P(s_t)$ con le probabilità Π_t del MCTS. I pesi θ_{it} della rete vengono allenati tramite *gradient descent* su una funzione di costo l che somma l'errore quadratico medio (MSE) e *Cross-entropy*: $l = (z_t - v(s_t))^2 + \Pi_t \log(P(s_t))$. . Vengono generati così dei nuovi pesi θ_{it+1} e si inizia l'iterazione successiva.

3. Metodi

3.1 Policy iteration

L'iterazione di policy è un algoritmo classico che genera una sequenza di policy che vanno via via migliorandosi, alternando tra *valutazione,* stimando la "bontà" di una policy, e *miglioramento*, generando una nuova policy aggiornata. L'algoritmo di selfplay di AlphaZero può essere visto nel suo insieme come un algoritmo di *policy iteration*, dove il MCTS viene usato sia per valutare sia per migliorare la policy corrente.
Il miglioramento si ha quando viene eseguito il MCTS che esegue la ricerca andando a migliorare le mosse scelte dalla rete, mentre la valutazione della policy corrente viene fatta confrontando le probabilità e il q-value predetto dalla rete con le probabilità prodotte dal MCTS e il vincitore del gioco.
Dopo il miglioramento e la valutazione si può passare all'iterazione successiva usando la policy aggiornata.

3.2 Conoscenze di partenza

L'obiettivo dell'algoritmo di AlphaZero è quello di raggiungere alte prestazioni nel proprio dominio partendo da tabula-rasa, senza nessun elemento della conoscenza umana, avendo solamente le regole del gioco. Questo rende l'algoritmo utilizzabile in giochi e contesti eterogenei tra loro: dal gioco del Go, agli Scacchi e in pratica a qualsiasi gioco totalmente osservabile. Le logiche dell'algoritmo che dipendono dal dominio sono le seguenti:

- Le regole del gioco o del dominio in questione. Queste regole infatti vengono utilizzate nelle ricerche del MCTS per simulare un'azione e per dare un valore ad una partita se essa è finita.
- L'input *feature* della rete neurale. Nel caso del Go l'input è rappresentato dalla scacchiera 19x19, quindi il primo layer della rete sarà modellato di conseguenza.

- Come accrescere i dati di training (opzionale). Le regole di un gioco potrebbero essere invarianti rispetto a determinate trasformazioni. Ad esempio le regole del Go non cambiano se lo stato del gioco viene ruotato oppure riflesso. Questa invarianza può essere sfruttata per generare nuovi dati di selfplay su cui allenare la rete.

3.3 Monte Carlo Tree Search

In questo paragrafo viene descritto nel dettaglio il funzionamento del Monte Carlo Tree Search.

Ogni nodo s nell'albero contiene degli archi (s, a_i) ognuno per ogni azione a_i valida. Ogni arco memorizza al suo interno una serie di variabili:

$$\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$$

dove *N(s,a)* è il contatore di visite, *W(s,a)* è il q-value totale, *Q(s,a)* è il q-value medio e *P(s,a)* è la probabilità a priori di selezionare l'arco.

La ricerca dell'albero itera sulle prime tre fasi descritte in seguito, dopo di che seleziona la mossa da eseguire:

1) Selezione. La prima fase inizia dalla radice dell'albero di ricerca e termina quando la simulazione raggiunge un nodo s_L foglia, ovvero un nodo ancora mai visitato, allo step L. Per ogni step $t<L$ un arco (cioè un'azione) a è selezionato in questo modo:

$$a = argmax_a(Q(s_t,a) + U(s_t,a))$$

dove

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} .$$

Il parametro c_{puct} è una costante che determina il livello di esplorazione dell'albero. In questo modo nelle prime simulazioni verranno selezionate

azioni con alta probabilità a priori e basso numero di visite, ma successivamente si tenderà verso azioni con maggior q-value.

2) Espansione e valutazione. Una volta che la fase di selezione raggiunge un nodo foglia s_L essa termina, e il nodo viene valutato dalla rete neurale $(P, v) = f_\theta(s_L)$:

- Il nodo viene espanso e ogni arco (s_L, a_i) è inizializzato con:

$$\begin{aligned} N(s_L, a_i) &= 0 \\ W(s_L, a_i) &= 0 \\ Q(s_L, a_i) &= 0 \\ P(s_L, a_i) &= P_i \end{aligned}$$

  dove P_i è la componente del vettore P ritornato dalla rete che indica la probabilità (a priori) di scegliere l'azione a_i.
- Il valore v viene ritornato al nodo precedente nella fase successiva.

3) Backup. Le statistiche di ogni nodo incontrato durante la fase di selezione vengono aggiornate a ritroso in ogni step $t \leq L$:

- Il contatore delle visite viene incrementato $N(s,a) = N(s,a)+1$,
- viene sommato al q-value totale W il termine v ritornato dalla rete nella fase di valutazione $W(s_t,a) = W(s_t,a)+v$,
- viene ricalcolato il q-value medio come q-value totale diviso il numero di visite $Q(s_t,a) = W(s_t,a) / N(s_t,a)$.

4) Selezione della mossa. Dopo aver iterato a sufficienza sulle prime tre fasi (*1600* iterazioni in AlphaGo Zero) il MCTS seleziona una mossa da giocare ritornando un vettore di probabilità. Ogni componente è proporzionale al suo numero di visite esponenziato:

$$\Pi_i = \frac{N(s_0, a_i)^{1/\tau}}{\sum_b N(s_0, b)^{1/\tau}}$$

dove τ è un parametro che controlla la temperatura. Una volta che il MCTS ha selezionato una mossa da giocare l'albero *viene riutilizzato*: il nodo figlio

corrispondente all'azione scelta diventa la nuova radice, mentre il resto dell'albero viene scartato. Inoltre è possibile sommare al vettore un rumore artificiale, in modo da poter esplorare nuove mosse non considerate dalla ricerca dell'albero.

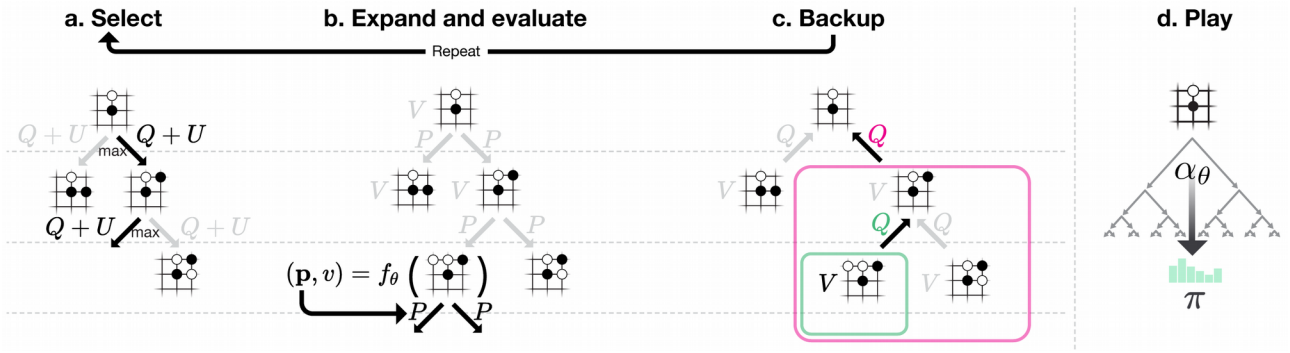

Figura 4: Monte Carlo Tree Search in AlphaZero: ***a.*** *Selezionare la mossa che massimizza il q-value Q sommato al valore U, dipendente dalla probabilità a priori e dal numero di volte in cui la mossa è già stata selezionata. Si seleziona una mossa ricorsivamente fino a che non si arriva ad uno stato foglia s_L, ovvero uno stato ancora non espanso* ***b.*** *Lo stato s_L viene valutato dalla rete, che ritorna v e P. Ogni componente P_i viene salvata negli archi corrispondenti alle mosse eseguibili in s_L* ***c.*** *Vengono ritornati i q-values di ogni mossa selezionata ricorsivamente* ***d.*** *Viene eseguita la mossa che massimizza il numero di visite. Immagine tratta da referenza (2)*

3.4 Rapporto esplorazione sfruttamento

Nell'algoritmo di ricerca è posta particolare attenzione ai livelli di esplorazione dell'albero e sfruttamento delle mosse conosciute. Infatti se non si facesse abbastanza esplorazione la policy non avrebbe possibilità di miglioramento, se invece non si sfruttassero le mosse allora la policy non potrebbe avere alte prestazioni. I parametri che controllano questo rapporto sono inseriti nel MCTS, e sono c_{puct} e τ :

- $c_{puct}>0$ controlla l'esplorazione dell'albero di ricerca durante la fase di selezione, quindi all'interno dell'albero. Un c_{puct} troppo alto favorirebbe la selezione di mosse che sono state visitate poco e con alta probabilità a priori, mentre un c_{puct} troppo basso sfrutterebbe solo le mosse con un q-value elevato e che quindi già simulate in precedenza;

- τ controlla la temperatura del vettore Π al di fuori dell'albero di ricerca. Infatti τ interviene dopo le iterazioni di ricerca, nella fase finale di selezione della mossa.

3.5 Struttura della rete neurale di AlphaGo

Per il training di AlphaGo è stata utilizzata una rete neurale costituita da un singolo layer convoluzionale seguito da 19 o 39 *residual blocks* (5). Il layer convoluzionale è formato da:

1. Convoluzione di 256 filtri 3x3 con stride 1;
2. Batch normalization;
3. RELU come funzione di attivazione.

Ogni residual block invece applica al suo input i seguenti layer:

1. Convoluzione di 256 filtri 3x3 con stride 1;
2. Batch normalization;
3. RELU;
4. Convoluzione di 256 filtri 3x3 con stride 1;
5. Batch normalization;
6. Una connessione all'input di un blocco successivo;
7. RELU.

A questo punto l'output corrente viene passato a due diverse sezioni, una per calcolare le probabilità della policy e l'altra per calcolare il q-value corrente.

Per la policy si ha:

1. Convoluzione di 2 filtri 1x1 con stride 1;
2. Batch normalization;
3. RELU;
4. Un fully connected layer che ritorna in output un vettore di dimensioni 19^2+1 proporzionale alle probabilità delle azioni da scegliere (19^2 mosse più il passo).

Per il q-value invece:

1. Convoluzione di 1 filtro 1x1 con stride 1;

2. Batch normalization;
3. RELU;
4. Un fully connected layer connesso ad un hidden layer di dimensioni 256;
5. RELU;
6. Un fully connected layer connesso ad uno scalare;
7. *tanh* funzione di attivazione con l'output normalizzato su [-1, 1].

4. Reti neurali e deep learning

Il *deep learning* è un campo del machine learning che si basa su diversi livelli di rappresentazione dei dati, corrispondente a gerarchie di caratteristiche (o *features*) dei fenomeni osservati. Tra le architetture del deep learning, le più usate sono le reti neurali o *neural networks*.

Una rete neurale è un modello matematico ispirato al cervello biologico animale. Questi sistemi imparano a risolvere problemi partendo da esempi dati, senza bisogno di essere programmati in anticipo con regole che dipendono dal compito da svolgere. Ad esempio una rete neurale può imparare a riconoscere i volti delle persone dopo essere stata allenata con immagini etichettate “Si” o “No” a seconda se in quella data immagine era presente un volto oppure no.

Una neural network è costituita da neuroni artificiali connessi, come accade in un cervello, dove ognuno di essi, quando riceve un input, processa il proprio output e lo segnala ai neuroni successivi. Neuroni e connessioni spesso hanno un peso che varia il modo in cui processano il segnale in input e solitamente va ad aggiustarsi durante il training. Le neural networks sono ampiamente usate in campi come la *computer vision*, riconoscimento vocale e diagnosi mediche.

4.1 Funzionamento

Un neurone *j*, nel ricevere un input al tempo *t* dai neuroni predecessori, calcola il proprio output in funzione di:

- valori in input ovvero l'output o_{j-1} dei neuroni precedenti al livello *j-1*;
- un valore detto *bias*, che non dipende dal layer della rete e viene modificato solo durante l'apprendimento;
- una funzione che combina il *bias* e l'input del neurone;
- una funzione di attivazione ρ non lineare che calcola l'output $o_i(t)$.

Nel caso di neuroni di input, il valore iniziale viene dato dall'input del problema dato, mentre i neuroni di output forniscono l'output dell'intera rete. Un neurone i può essere collegato ad un neurone j tramite la connessione (o arco) w_{ij} a cui viene assegnato un peso.

4.2 Feed forward

Il passo forward della rete è la funzione che calcola l'output della rete a partire dai neuroni di input, e solitamente ha questo svolgimento:

- Si calcola l'input $p_j(t)$ del neurone j a partire dagli output dei neuroni precedenti $o(t-1)$ facendo una somma pesata con il weight delle connessioni
- Se il bias è presente si somma a p_j ottenendo net_j;
- Si calcola il valore della funzione di attivazione $\rho(net_j)$ e se ne restituisce l'output o_j.

4.3 Funzione di errore

La funzione di errore è una funzione che prende in input l'uscita della rete e il valore atteso e calcola il costo associato a questi valori. Un esempio di funzione di costo è la distanza Euclidea tra l'uscita della rete y e il valore atteso $\hat{y}$:

$$E(y, \hat{y}) = \frac{1}{2} \| y - \hat{y} \|$$

con derivata:

$$\frac{\partial E}{\partial \hat{y}} = \hat{y} - y$$

La derivata della funzione di errore sarà utile durante la fase di backpropagation.

4.4 Backpropagation

Una neural network come modello matematico è teorizzata fin dal 1943, non riscuotendo molto scalpore nella comunità scientifica, almeno fino al 1975 quando Paul Werbos presentò l'algoritmo della backpropagation che ne permetteva l'apprendimento automatico tramite *gradient descent* da una funzione di costo. Tuttavia la mole di calcoli necessaria per far funzionare l'algoritmo lo resero inutilizzabile fino a che la backpropagation non fu implementata su dispositivi per il calcolo parallelo come le GPU.

L'algoritmo consiste nel partire da un certo input di cui si conosce l'output atteso, calcolarsi l'uscita della rete e propagare l'errore all'indietro, calcolandone la derivata rispetto ad ogni peso e aggiornando quest'ultimi. Per il calcolo delle derivate rispetto ad un peso generico w_{ij} la backpropagation si avvale della ***chain rule*** (applicata due volte):

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}$$

dove net_j è l'output del neurone j prima che sia applicata la funzione di attivazione e o_j è l'output attivato. La chain rule è di fondamentale importanza per la backpropagation, perché permette di riscrivere la derivata dell'errore in termini semplici da calcolare. Partendo dal termine più a destra:

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\sum_{k=1}^{n} w_{kj}o_k = \frac{\partial}{\partial w_{ij}} w_{ij}o_i = o_i$$

Se il neurone si trova nel layer di input, o_i sarà semplicemente x_i.

Il termine centrale calcola la derivata dell'output attivato rispetto all'output non attivato. A titolo di esempio viene utilizzata la sigmoide come funzione di attivazione, caratterizzata da:

$$\rho = \frac{1}{1+e^{-x}}$$

con derivata:

$$\frac{d\rho}{dx}(x) = \rho(x)(1-\rho(x)) \cdot$$

Si ha quindi:

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j}\rho(net_j) = \rho(net_j)(1-\rho(net_j)) \cdot$$

Il valore del primo fattore a sinistra dipende se il neurone si trova nell'output layer oppure no. In caso positivo si ha che $o_j = y$ e quindi:

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}(\hat{y}-y)^2 = y-\hat{y}$$

Se si tratta invece di un neurone interno l'errore dipenderà dagli output di ognuno dei neuroni precedenti k e lo si può calcolare ricorsivamente:

$$\frac{\partial E}{\partial o_j} = \sum_k \left(\frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial o_j}\right) = \sum_k \left(\frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial net_k}w_{jk}\right)$$

Unendo i vari fattori si ha l'espressione:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_j, \quad \text{con } \delta_j = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j-\hat{y_j})o_j(1-o_j), & j \in output \\ (\sum_k w_{jk}\delta_k)o_j(1-o_j), & j \notin output \end{cases}$$

Una volta calcolati tutti i fattori necessari si può procedere all'aggiornamento dei pesi, scegliendo prima un learning rate $\eta > 0$ che andrà a modificare il modulo del vettore del gradient descent, ovvero quanto sarà modificato ogni peso. La formula seguente garantisce la diminuzione dell'errore sfruttando il segno della derivata, quindi in entrambi i casi in cui un aumento o diminuzione del peso generi un aumento o diminuzione dell'errore :

$$\Delta w_{ij} = -\eta\frac{\partial E}{\partial w_{ij}} = -\eta\delta_j o_i$$

5. Tecnologie

5.1 Tensorflow

Tensorflow è una libreria open source sviluppata originariamente da Google per la computazione efficiente di grandi moli di lavoro. La sua architettura permette la distribuzione e parallelizzazione dei calcoli su piattaforme diverse, come CPUs, GPUs che sfruttano nVidia CUDA e le nuove TPUs, acceleratori hardware creati per Tensorflow, ma anche smartphones Android o iOS, su un singolo computer o su un cluster. Il machine learning, e in particolare nel deep learning, richiede una grande potenza di calcolo, e Tensorflow è diventata una delle librerie più usate nel settore, usata in ambienti di produzione in aziende come Intel, Twitter e eBay.

5.1.1 Come funziona

Tensorflow permette di creare dei *dataflow graphs*, strutture che descrivono come i dati si muovono all'interno di un grafo.
Ogni nodo del grafo rappresenta un'operazione matematica, e ogni arco tra due nodi è un array multidimensionale, o *tensore*. Tensorflow fornisce questi strumenti esponendo delle funzioni in Python, che consente allo sviluppatore di scrivere il proprio software in un linguaggio di alto livello. Tuttavia l'esecuzione dell'applicazione viene eseguita da file binari scritti in *C++*, in questo modo lo sviluppatore ha a disposizione la flessibilità di un linguaggio ad alto livello pur mantenendo efficienza e velocità.

5.2 Keras

Keras è una libreria che espone API di alto livello nel linguaggio Python per sperimentare e prototipare velocemente deep neural networks.

Keras non è un framework per il machine learning, ma bensì un'interfaccia, svolgendo il ruolo di “frontend” sulle librerie per il calcolo ottimizzato come Tensorflow, Theano o Microsoft Cognitive Toolkit (CNTK). Le funzioni che offre sono più user-friendly e più specifiche rispetto alle API di Tensorflow, semplificando lo sviluppo di reti neurali. Keras, fungendo da interfaccia, non limita lo sviluppo ad un solo ecosistema, infatti si può allenare un modello di machine learning in un certo ecosistema e mandarlo in produzione sulla piattaforma desiderata, come iOS tramite CoreML, Google Cloud e perfino su un browser tramite codice Javascript accelerato con GPU e WebASM. Inoltre la libreria contiene tool utili per la manipolazione e il preprocessing di immagini e testo. Tutto ciò ha permesso a Keras di diventare una delle librerie di machine learning più diffuse al mondo e ad essere inclusa in Tensorflow dalla versione *1.4.0*.

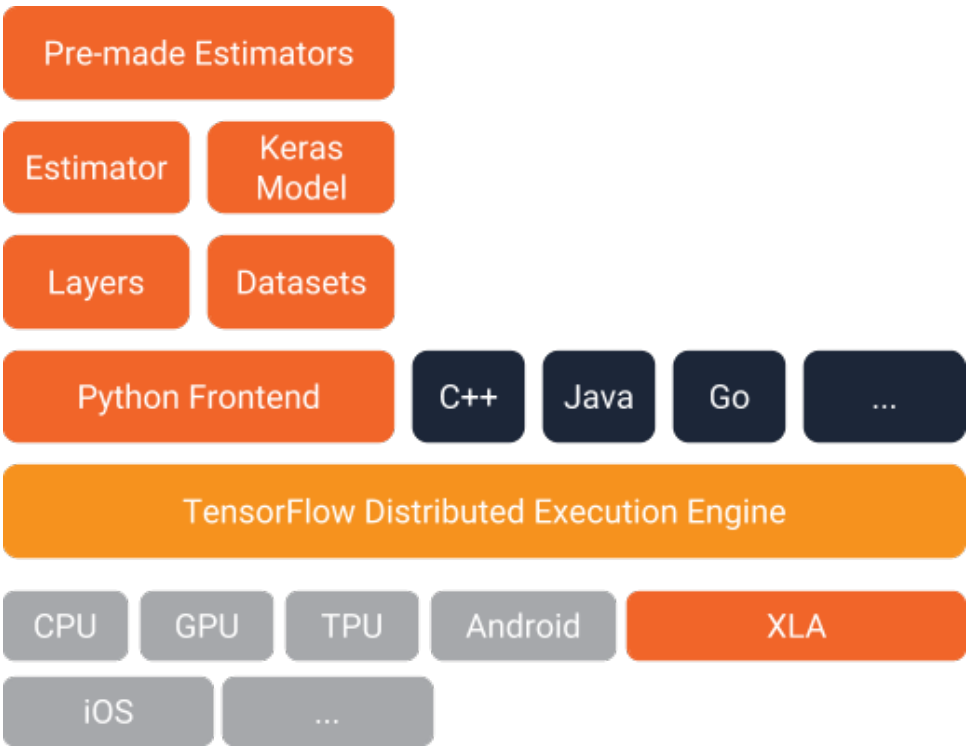

Figura 5: Struttura delle API di Keras. Immagine tratta da tensorflow.org

5.3 OpenAI gym

OpenAI gym è un framework pensato per lo sviluppo e il test di algoritmi di reinforcement learning. Non fa nessuna assunzione sull’agente da utilizzare

ed essendo scritto in Python è compatibile con librerie di calcolo come Tensorflow. Gym comprende numerosi ambienti di diversa difficoltà dove testare il proprio agente, dal *CartPole*, consistente nel bilanciare un palo sopra una base movente, ai giochi classici dell'Atari, come *Pacman* o *Space Invaders*, fino al controllo di agenti robot 2D e 3D. L'interfaccia che mette a disposizione permette di eseguire un'azione in un ambiente, passando così allo stato successivo, e mostrare una finestra grafica con lo stato del gioco. Non dando molte features particolari dell'ambiente, se non lo stato del gioco, *gym* è pensato per il reinforcement learning, nel quale l'agent deve trovare autonomamente il miglior modo di muoversi e di eseguire varie azioni.

6. Implementazione di AlphaZero

L’implementazione è stata realizzata completamente in Python, sfruttando librerie descritte nella sezione precedente. L’intero codice è disponibile su questa repository Github: github.com/MrSpadala/alpha-zero.
Si è cercato di velocizzare il più possibile il codice e al tempo stesso di mantenerlo pulito e semplice.

6.1 Struttura dei files

Viene riportata la descrizione e le specifiche dei file principali per il funzionamento dell’algoritmo.

6.1.1 train.py

Questo file contiene l'intera pipeline di training e le sue funzioni. La funzione *train(),* come si intuisce, lancia l'allenamento, consistente fondamentalmente in tre passi base:

Inizializzazione (passo 0): Cerca di riprendere lo stato del training se questo è stato interrotto. Questo viene effettuato tramite la chiamata *iteration = globals.get_last_iteration()* che ritorna il numero dell'iterazione corrente.

1. *Selfplay*: gioca un numero di partite contro sé stesso e una volta completate le memorizza nella variabile *examples*. Successivamente i dati di selfplay vengono espansi inserendo anche le varie simmetrie del dominio di interesse, con la chiamata *all_symmetries(examples)*. Infine i dati di selfplay vengono salvati su disco.

2. *Net training*: singola chiamata alla funzione *train_net(examples)* che allena la rete sui dati di selfplay memorizzati nella variabile *examples.*

3. *Net evaluation*: funzione *evaluate_net(net, net_opponent),* che esegue molteplici partite facendo giocare due network diverse, e ritorna la somma dei risultati (vedi *evaluate_net* più avanti). Se *score* è maggiore o uguale a zero allora il modello viene accettato, altrimenti viene scartato. In entrambi i casi il training prosegue con una nuova iterazione dal passo 1 dopo aver salvato su file l'iterazione corrente con la chiamata a *update_iteration_file.*

Parallelizzazione

Sia la fase di selfplay sia quella di net evaluation sono parallelizzate su più cores in quanto sono molto CPU intensive. La parallelizzazione viene effettuata attraverso *multiprocessing.Pool,* incluso nella standard library di Python, creando un certo numero di processi (non thread) concorrenti. Il motivo di creare processi al posto dei thread è dovuto a ragioni di performance: thread concorrenti vengono eseguiti sul singolo processo Python usando un singolo core, mentre processi diversi possono sfruttare più cores.

A causa della parallelizzazione per ogni funzione *foo* che viene chiamata dalla training pipeline, esiste una funzione *foo_process*, ovvero la target function che eseguiranno i nuovi processi (ad esempio esiste la funzione *selfplay* e *selfplay_process*).

La parallelizzazione dei job adotta il paradigma master-slave: un processo principale che distribuisce i compiti e i processi worker che lo eseguono. Il lavoro di ogni slave consiste nell'eseguire *games_per_chunk* partite. All'inizio del programma vengono lanciati un numero di worker *parallel_num*, calcolando il numero di job da assegnare.

Quando un processo ha terminato il suo job verifica se ce ne sono ulteriori da svolgere leggendo banalmente il valore di un contatore condiviso tra i vari worker (in maniera sincronizzata) secondo lo schema seguente:

```
def worker_thread(counter, lock):

    loop forever:
        get lock
        counter ← counter – 1
        release lock

        if counter < 0:
            break
        play games
```

Nota: a causa di alcune limitazioni nella libreria Keras non è possibile importarla nel processo master e successivamente importarla nei processi figli, comporta infatti un deadlock (issue su github github.com/keras-team/keras/issues/9964). Per questo motivo se il processo master ha bisogno di operare su Keras, non può importare la libreria direttamente, ma lanciare un processo figlio ausiliario che importi la libreria e che svolga il compito.

Parametri

- *iters*: numero di iterazioni da svolgere
- *parallel_num*: numero di processi da lanciare in parallelo
- *num_self_games*: numero di partite da giocare per ogni iterazione
- *games_per_chunk:* numero di partite giocate alla volta da ogni processo worker
- *num_eval_games*: numero di partite da giocare durante la fase di evaluation
- *always_accept_model*: booleano, indica se accettare la nuova rete alla fine del training senza eseguire la fase di valutazione
- *evaluate_only*: booleano, indica se eseguire solo la fase di valutazione

- *show_moves*: booleano, indica se visualizzare la partita corrente sul terminale, per avere un riscontro visivo. Se è posto a True automaticamente *parallel_num* è settato ad 1.

Funzioni

- *selfplay()*: si occupa di far giocare partite parallelamente. Crea un contatore dei job che sarà condiviso dai workers *chunk_counter*, crea una coda anch'essa condivisa dove i processi scriveranno i propri risultati, e lancia i processi. Una volta terminate tutte le partite, ottiene dalla coda i dati di tutti i processi, nella forma:

  $$[[state_1, \pi_1, v_1], [state_2, \pi_2, v_2], \ldots, [state_n, \pi_n, v_n]]$$

  di dimensioni *(n, 3)*, dove *n* è il numero di mosse totali delle partite giocate. Quest'ultimo vettore viene ritornato.

- *selfplay_process(chunk_counter, games_per_chunk, all_moves)* funzione che esegue *games_per_chunk* partite alla volta usando una stessa rete come giocatori; essa viene caricata in memoria con la chiamata *net=NNet('best')*. Le partite vengono svolte finchè c'è un job disponibile, iterando sull'intervallo con la variabile *game_it*. Il numero di iterazione corrente viene usato come seed per generare lo stato iniziale di una partita tramite *generate_training_game(game_it)*, dove due seed uguali generano lo stesso stato (vedi *game.py*).
  Ogni mossa viene scelta chiamando *mcts.get_pi(player)* e scegliendo l'indice dell'elemento maggiore tramite *argmax*. La mossa scelta viene poi memorizzata in un vettore: *[state, pi, 0],* dove lo *state* è lo stato della partita dal punto di vista del giocatore corrente (vedi *canon_board* in *game.py*), *pi* è il vettore di probabilità ritornato dal MCTS, e 0 è un placeholder che andrà aggiornato con il vincitore della partita una volta finita.
  Terminata un partita, le mosse salvate vengono inserite nella coda condivisa *all_moves*.

- *train_net(examples)* e *train_net_process(examples):* Crea un nuovo processo che esegue il training della rete sui dati presenti in *examples*. Una volta terminato il training viene salvato il checkpoint della rete.

- *evaluate_net(net, net_opponent)*: Prepara i processi che eseguiranno partite facendo giocare *net* contro *net_opponent.* I processi vengono parallelizzati allo stesso modo di della funzione *selfplay().* Ritorna uno scalare che rappresenta la somma dei punteggi di tutte le partite giocate, dove ogni vittoria di *net* vale +1, ogni vittoria di *net_opponent* vale -1 e il pareggio vale 0.

- *evaluate_net_process(net, net_opponent)*: Esegue *games_per_proc* partite analogamente a *selfplay_process*, ritorna la somma dei risultati delle partite.

6.1.2 MCTS.py

Il file contiene la logica del Monte Carlo Tree Search guidato da una neural network. L'albero di ricerca è rappresentato in memoria come un albero di stati (archi), dove ognuno mantiene una propria lista di stati figli. Uno stato padre avrà quindi uno stato figlio per ogni mossa valida che si può eseguire. Per accedere e traversare l'albero è quindi sufficiente usare la sua radice e spostarsi di stato padre in figlio.

Ogni arco è rappresentato da: lo stato della partita, il numero di volte che è stato visitato, il q-value totale, il q-value medio, la probabilità a priori di essere scelto dall'arco padre, una lista di archi figli e una lista di mosse valide che si possono eseguire a partire dallo stato corrente.
Si noti che lo stesso albero di ricerca è usato per i due avversari, in questo modo si ottimizza la computazione ottenendo un numero doppio di simulazioni del MCTS rispetto al caso di due alberi distinti. Ogni stato simulato (figlio) è generato eseguendo una mossa di un certo giocatore in uno stato padre, quindi per ogni livello di profondità dell'albero si ha un'inversione del

giocatore corrente. Questa proprietà è sfruttata per eseguire la ricerca sullo stesso albero dai punti di vista di entrambi i giocatori.
Inoltre, sempre per ottimizzare la computazione, non si elimina l'albero dopo l'esecuzione di una mossa da parte di un giocatore. Così facendo si perderebbero i dati riguardanti i livelli inferiori e bisognerebbe iniziare una nuova ricerca da zero alla prossima mossa. Invece, si può conservare la parte dell'albero corrispondente alla mossa che è stata scelta “scalando” l'albero in modo da far diventare lo stato corrente (lo stato che si ha dopo la mossa eseguita dal giocatore) la radice dell'albero da usare nella prossima ricerca.

Il Monte Carlo Tree Search utilizzato da AlphaZero varia dalla versione “classica” in quanto non si eseguono dei rollout sui nuovi stati, ma si usa una neural network per valutare il q-value e le probabilità (a priori) di scegliere ogni mossa. Questo avviene durante la fase di espansione, cioè quando la ricerca incontra uno stato foglia che ancora non è stato esplorato; il MCTS originale in questo caso compierebbe un rollout (o playout) seguendo una rollout policy (che potrebbe essere anche eseguire mosse random) fino a che la partita termina. Il MCTS di AlphaZero invece, una volta che incontra uno stato foglia, usa la neural network per valutare la posizione corrente. Questo è un grande vantaggio nei confronti del MCTS classico in quanto la rollout policy nel corso del training migliora sempre di più e fornisce probabilità a priori sempre più “potenti”. Affinché questo metodo funzioni ovviamente è necessario eseguire centinaia se non migliaia di simulazioni per ogni valutazione delle mosse, a seconda della complessità del gioco o del dominio di interesse.

Per iniziare il Monte Carlo Tree Search dallo stato radice dell'albero viene chiamata la funzione *MCTS.get_pi,* che per il numero di simulazioni scelto chiamerà la funzione *MCTS.search*, che implementa la ricerca vera e propria in maniera ricorsiva. Una volta terminate tutte le simulazioni la funzione ritorna un vettore di dimensioni uguali a quello dello spazio del gioco (ad esempio nel gioco del Go è possibile fare 19x19 mosse più passare) dove ogni componente *i*-esima del vettore vale quanto la probabilità di scegliere la mossa *i*-esima.

Scelte di implementazione

L'albero di ricerca è memorizzato come una un albero di stati ed essi sono rappresentati in memoria dalla classe *Edge*. Ognuno si memorizza (oltre alle proprietà descritte precedentemente) il proprio stato, un vettore booleano di mosse valide *valids*, dove *valids[a]==1* se l'azione *a* è consentita dalle regole del gioco, e una lista degli stati figli, che viene inizializzata vuota.

L'albero di ricerca è a sua volta rappresentato dalla classe *MCTS* memorizzandosi la radice dell'albero e la rete neurale con cui effettuare le evaluations, inoltre tiene un contatore delle mosse eseguite.

Parametri

- *sim_per_move*: quante simulazioni del MCTS da eseguire per scegliere ogni mossa;
- *cpuct*: parametro che aumenta linearmente la possibilità di selezionare un certo stato figlio visitato poche volte rispetto ai suoi fratelli;
- *noise_multiplier*: livello del rumore artificiale aggiunto al vettore π. Il rumore in questione è una distribuzione di probabilità random moltiplicata per *noise_multiplier* e successivamente sommato a π;
- *tau*: parametro che controlla la temperatura delle probabilità finali dentro Π ;
- *tau_moves*: quante mosse eseguire prima di modificare tau.

Funzioni

- MCTS.get_pi(player): ritorna un vettore di probabilità delle mosse dal punto di vista del giocatore player. Dopo che le simulazioni sono terminate costruisce il vettore pi basandosi sul parametro tau di temperatura. Se tau è uguale a zero la temperatura è al suo minimo: tutte le componenti di π saranno azzerate tranne l'unica corrispondente alla mossa migliore selezionata durante la visita dell'albero, che sarà pari a 1. Viceversa, se tau è maggiore di zero, saranno presenti nel vettore anche le probabilità delle mosse non selezionate. Solitamente all'inizio della partita tau è impostata alta e va a decrescere durante il resto della partita.

Prima di ritornare il vettore π, si crea temporaneamente una distribuzione di probabilità random, un rumore artificiale, che andrà a sommarsi a Π . In questo modo ci si assicura che tutte le mosse potrebbero essere scelte.

- *MCTS.search(edge, player)*: implementa la ricerca dell'albero vera e propria in maniera ricorsiva che viene svolta dal punto di vista del giocatore *player* a partire dallo stato *edge*.
  Inoltre la funzione esegue una singola simulazione del MCTS, per effettuarne di più basta chiamare la stessa iterativamente sullo stesso albero.

  Segue lo pseudo-codice per chiarezza:

```
def search(edge, giocatore):
    if edge.state is game over:
        {N, W, Q} ← {N+1, W+winner, W/N}
        return -winner

     if edge is leaf:
        {pi, v} ← predict state with net
        {edge.children} ← new children states
        return -v
    else:
        {Q + U values} ← vector of Q + U values of children
        {action} ← argmax(Q + U values)
        {v} ← search(edge.children[action], -player)
        {N, W, Q} ← {N+1, W+v, W/N}
        return -v
```

dove gli U values degli stati figli sono calcolati secondo la formula:

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)}$$

e dipendono linearmente dal parametro c_{puct}.

Segue lo pseudo-codice del MCTS completo:

```
def get_pi(player):
    for sim_per_move times do:
        search(root, player)
    {visits} ← vector of visits count of children
    {visits} ← visits + (random noise) * (noise_multiplier)
    {pi} ← adjust visits according to temperature tau
    {action} ← argmax(pi)
    {root} ← root.children[ action ]
    return pi
```

6.1.3 game.py

Il file contiene la logica del gioco o del dominio generale su cui si vuole applicare AlphaZero. Le funzioni da implementare con la relativa documentazione sono presenti nella classe *GameState*:

- *action_size*: ritorna il numero di azioni possibili;
- *valid_moves*: ritorna un vettore booleano, dove per ogni mossa *i*-esima *valid_moves[i]=1* se è valida;
- *canon_board(player)*: ritorna il campo di gioco dal punto di vista del giocatore *player*. La rete neurale infatti può solo predire le mosse come se stesse giocando per uno solo dei due giocatori. A seconda del giocatore la funzione ritorna il campo da gioco originale oppure invertito;
- *copy*: ritorna una copia dello stato di gioco;
- *get_winner*: ritorna 1, -1 oppure 0 rispettivamente se ha vinto il giocatore 1, se ha vinto il giocatore -1, se non ha vinto ancora nessuno. Se la partita non è ancora terminata oppure si è conclusa con un pareggio la funzione ritorna 0 in entrambi i casi;

- *next_state(action, player, copy=False)*: esegue l'azione *action* sul campo di gioco da parte del giocatore *player*. Se l'argomento opzionale *copy* è False la mossa viene eseguita sullo stesso campo di gioco (side effect), altrimenti viene ritornata una nuova copia dello stato di gioco;
- *all_symmetries(examples)*: sia *examples* una lista nella forma *[[state1, pi1, v1], [state2, pi2, v2], ...]*, la funzione ritorna una nuova lista che contiene gli elementi della lista precedenti più le eventuali copie specchiate o rotate sfruttando le simmetrie del gioco;
- *generate_training_game(iteration)*: genera un nuovo stato di gioco per l'iterazione corrente. Il parametro *iteration* può essere visto come un seed, due stati generati dallo stesso seed saranno uguali.

6.1.4 Nnet.py

Questo file contiene la struttura della rete neurale e qualche utility. Il modello della neural network è composto da:

1. Convoluzione di 256 filtri 4x4 con stride 1;
2. Batch normalization;
3. RELU come funzione di attivazione;
4. 3 layers di convoluzione con 256 filtri 3x3 con stride 1, Batch Normalization e RELU;

A questo punto la rete si biforca per l'output del vettore pi e dello scalare v.

Per il vettore della policy si ha:

1. Convoluzione di 2 filtri 1x1 con stride 1;
2. Batch normalization;
3. RELU;
4. Un fully connected layer con funzione di attivazione softmax che ritorna in output un vettore di dimensioni uguali alle mosse possibili del gioco.

Per il q-value invece si ha:

1. Convoluzione di 1 filtro 1x1 con stride 1;
2. Batch Normalization;

3. RELU;
4. Un fully connected layer di dimensioni 256;
5. RELU;
6. Un fully connected layer connesso ad uno scalare;
7. *tanh* funzione di attivazione con l'output normalizzato su [-1, 1].

La rete usa come funzione di errore la somma della Mean Sqared Error e Cross Entropy.

7. *Training*

E' stato effettuato il training con AlphaZero di una policy per il gioco di forza quattro. Si noti che il codice e la logica dell'algoritmo è invariante rispetto al gioco o al dominio di applicazione, se non per il codice nel file *game.py* che ne implementa le regole.

7.1 Difficoltà riscontrate

7.1.1 Limitazioni GPU

A seconda se ci si trovi nella fase di selfplay o di apprendimento, la GPU del calcolatore può essere nascosta volontariamente a Keras e Tensorflow. Più precisamente è nascosta durante le fasi di selfplay e di evaluation (passo forward della rete neurale) ed è visibile durante il training (gradient descent). Questa scelta è dovuta per ragioni di performance: usando la GPU durante il selfplay, ogni worker in parallelo avrebbe dovuto importare la propria copia del modello della rete Keras all'interno della VRAM della GPU. Purtroppo questa soluzione è impraticabile poiché ogni copia del modello occupa ~2.6GB di VRAM e quest'ultima andrebbe a saturarsi istantaneamente. Allora ho progettato un'altra soluzione: un processo master su cui gira Tensorflow/Keras con GPU visibile e processi slaves che in maniera sincrona mandano i job da far eseguire alla scheda grafica. Dopo aver eseguito vari test, i risultati hanno mostrato il bottleneck della scheda grafica: ognuno dei 4 worker paralleli ha registrato un ritardo medio di ~0.015s tra il momento in cui ha messo in coda un job e il momento in cui ha ricevuto indietro i risultati. Provando ora a rendere invisibile la GPU e a lavorare solo con la CPU usando 4 workers indipendenti, quindi senza bisogno di sincronizzazione, si ottiene un tempo medio di ~0.0055s.

Considerando ora una media di 140 valutazioni della rete per mossa, 32 mosse per partita e 1500 partite divise tra 4 workers, si ottiene la differenza di tempo tra la soluzione con e senza GPU per una singola iterazione:

$$\Delta t = (0.015 - 0.0055)\frac{secondi}{valutazione}\,140\frac{valutazioni}{mossa}\,32\frac{mosse}{partita}\,375\,partite \approx 4\text{h } 30\text{m}$$

7.1.2 Ottimizzazione iperparametri

La scelta intelligente degli iperparametri del training è risultata un'operazione molto complessa, attraverso un costante *trial and error*. Essi non comprendono esclusivamente gli iperparametri della rete neurale, come learning rate o il numero di epochs, ma anche il numero di simulazioni del MCTS per ogni mossa, il valore di c_{puct} che controlla il livello di esplorazione dell'albero, la percentuale del rumore artificiale per ogni mossa, etc. Ogni volta che si modifica uno di questi il training può cambiare radicalmente, in bene o in male, quindi è necessario fare dei test che consistono fondamentalmente nell'eseguire delle iterazioni e valutare l'impatto che hanno avuto sul training. Il problema principale è stato il tempo necessario per svolgere una singola iterazione, nell'ordine di una decina di ore, a cui va aggiunto il tempo per l'effettiva valutazione del training.

Gli iperparametri sono stati cambiati molto di frequente durante le iterazioni, cercando di favorire l'apprendimento della rete:

- Iterazione *0~10*: Fase iniziale del training, il numero di simulazioni del MCTS è 400 e non viene aggiunto nessun rumore casuale. Inoltre al termine di ogni iterazione la rete non viene valutata per decidere se scartarla o meno, ma viene accettata a prescindere. Questo ha favorito l'apprendimento delle basi di gioco.
- Iterazione *10~45*: Fase intermedia del training, il numero di simulazioni è stato aumentato a 800, portando il tempo di esecuzione di un'iterazione a più di 12h, e si è iniziato ad aggiungere un rumore casuale alla scelta della mossa. Ciò assicura che tutte le mosse potrebbero essere scelte durante le partite.

- Iterazione *45~60*: Si è alzato il valore del parametro c_{puct} per dare più peso alle probabilità a priori di ogni mossa. Inoltre si è abbassato il learning rate cercando di ottimizzare la rete pur cercando di evitare di fare overfit sui dati di gioco.

7.2 Dati tecnici

7.2.1 Hardware utilizzato

- CPU: Intel i5-4670 @ 3.4 GHz, quad-core;
- GPU: nVidia GTX 780, 3GB di VRAM e 2304 Cuda cores;

7.2.2 Dati del training

- Iterazioni svolte totali: *86*
- Iterazioni svolte e accettate: *60* (*~30%* di policy scartate)
- Numero partite svolte: $1{,}68 \times 10^5$
- Numero totale mosse salvate: $5{,}02 \times 10^6$
- Tempo totale di elaborazione: *~50* giorni
- Numero di partite di selfplay per iterazione: *1500*
- Tempo richiesto per iterazione: *7h* con *400* simulazioni del MCTS per mossa, *12h 30m* con *800* simulazioni del MCTS per mossa
- Tempo medio per mossa: *~3s*
- Tempo di elaborazione * core: *~4800h*

7.3 Risultati del training

Il training è continuato per circa 50 giorni, comprendendo il tempo di selfplay e valutazione delle policy.

7.3.1 Osservazioni

- Iterazione *0~10*: la policy gioca in maniera pseudo-random, cercando raramente di eseguire un attacco oppure di difendersi.

- Iterazione *10~25*: la policy ha imparato le basi del gioco eseguendo attacchi basilari quando si presenta l'occasione di posizionare quattro pedine in fila con una singola mossa. Queste occasioni per vincere la partita vengono sfruttate non sempre ma la maggior parte delle volte, tuttavia ancora non ha sviluppato capacità di difesa: quando l'avversario è sul punto di mettere la quarta pedina, vincendo la partita, la rete non cerca di bloccare la futura mossa.

- Iterazione *25~45*: gli attacchi basilari vengono ora eseguiti sempre, e ha iniziato a mostrare capacità di difesa, bloccando alle volte le mosse vincenti dell'avversario.

- Iterazione *45~60*: le capacità di difesa sono state perfezionate, riuscendo a bloccare ogni volta il posizionamento della quarta pedina da parte dell'avversario. A volte, ma non sempre, riesce anche a prevedere doppi giochi dell'avversario, bloccandoli in anticipo. Si è notato inoltre il miglioramento del posizionamento nel campo di gioco, cercando di non giocare le proprie pedine in maniera troppo dispersiva, ma mantenendo una certa compattezza. Manca ancora tuttavia una strategia di attacco, come ad esempio effettuare doppi giochi.

7.3.2 Risultati

Viene riportato l'errore di *(P, v)* della rete ad ogni iterazione rispetto ai valori del MCTS, valutato sui dati di selfplay generati dalla policy all'iterazione *60*:

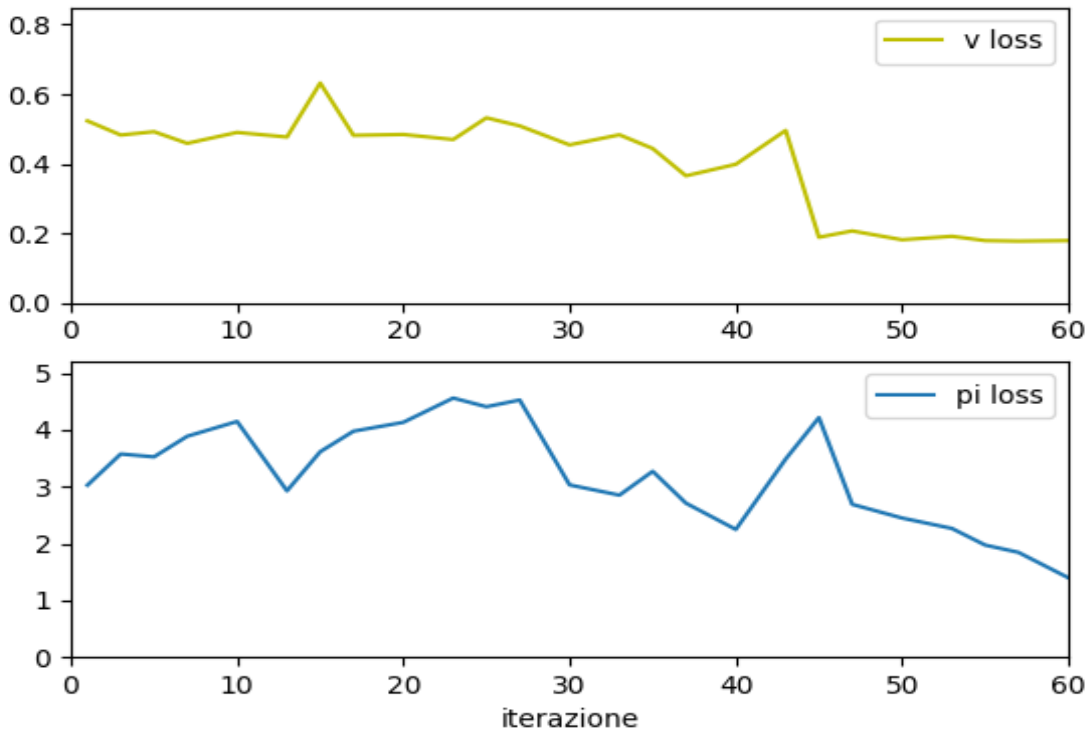

Figura 6: Andamento dell'errore sui dati di selfplay generati dalla policy migliore. L'errore decrescente indica l'apprendimento dei valori del MCTS da parte della rete.

Per valutare il livello di gioco è stata fatta giocare la policy migliore contro alcune reti delle iterazioni precedenti:

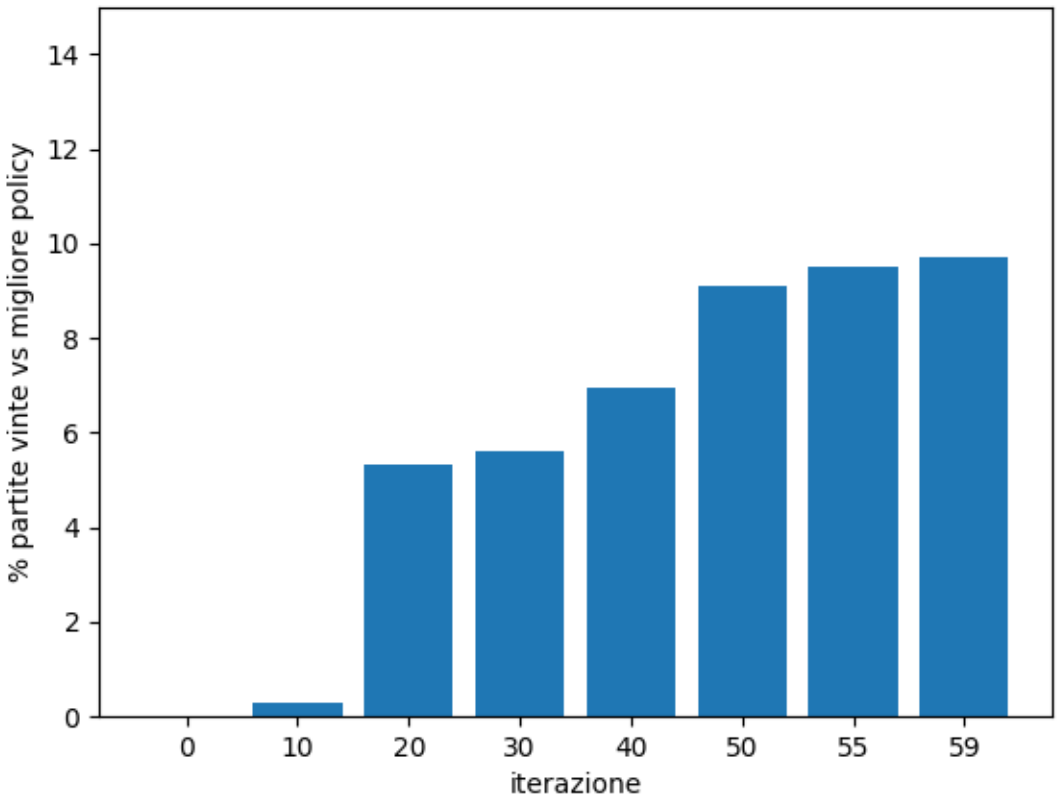

Figura 7: Il grafico riporta le vittorie (in percentuale) delle reti contro la policy migliore. Si nota il miglioramento del livello di gioco andando avanti con le iterazioni. I test sono stati eseguiti con 1400 simulazioni del MCTS per mossa senza l'aggiunta di rumore artificiale, in modo da eseguire le migliori mosse possibili.

8. Limitazioni e miglioramenti futuri

Anche se uno dei successi di AlphaZero è stato allenare una policy per Go con pochi giorni di training (*1*), bisogna considerare che l'hardware a disposizione comprendeva più di *5000* TPUs. L'enorme mole di calcoli effettuati da AlphaZero rende l'allenamento molto dispendioso in termini di tempo anche in contesti meno complessi del Go, come forza quattro, superando anche il mese di tempo (contando anche il tempo di valutazione delle policy e il tempo perso nel generare policy che verranno scartate). Il tempo di CPU è un fattore critico per il training di AlphaZero: la CPU risulta occupata per circa il *99%* del tempo totale della computazione contro l'*1%* della GPU usata solo durante il gradient descent. Un possibile miglioramento potrebbe essere riscrivere la logica del MCTS e del gioco in un linguaggio più efficiente di Python, per velocizzare la fase di selfplay.

Per cercare di garantire un allenamento uniforme si è cercato di far eseguire partite il più possibile eterogenee. Dato che, data una neural network e uno stato di una partita, la scelta di ogni mossa è deterministica (se non per la frazione di rumore artificiale che viene aggiunto), uno stesso stato iniziale del campo di gioco genererebbe uno stesso svolgimento della partita, usando tempo computazionale per generare dei dati già presenti. Per diminuire il più possibile il numero di duplicati, gli stati iniziali di una partita non sono vuoti, cioè senza nessuna mossa eseguita, ma sono partite già iniziate, solitamente con fino a *6* mosse totali posizionate (*3* per ogni giocatore). Stati iniziali di partenza diversi garantiscono un diverso svolgimento della partita e quindi dati di training eterogenei. Tuttavia, in questo modo la policy non ha la possibilità di allenarsi sulle prime mosse della partita, che potrebbero risultare piuttosto deboli se la partita viene iniziata da zero.

9. Conclusioni

AlphaZero dimostra che un algoritmo basato puramente sul reinforcement learning è realizzabile: è possibile allenare un’intelligenza artificiale, non necessariamente in maniera supervisionata attraverso esempi dell’uomo, anche nei contesti più complessi, partendo solamente dalle regole base del dominio. Allenare una policy attraverso AlphaZero significa avere prestazioni attese maggiori, rispetto ad un allenamento basato su dati di giocatori umani esperti. Tuttavia si deve considerare che la lunga fase di training richiede una grande potenza di calcolo che non tutti potrebbero avere a disposizione.

10. Referenze

1. D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan & D. Hassabis. *Mastering Chess and Shogi by Self-Play with a general Reinforcement Learning Algorithm* (2017)

2. D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel & D. Hassabis. *Mastering the Game of Go without Human Knowledge,* 2017.

3. Richard S. Sutton & Andrew Barto. *Reinforcement Learning: An Introduction,* 1998-2017.

4. Michael A. Nielsen, "*Neural Networks and Deep Learning*", Determination Press, 2015.

5. He, K., Zhang, X., Ren, S. & Sun, J. *Deep residual learning for image recognition*. In IEEE Conference on Computer Vision and Pattern Recognition, 770–778 (2016).

6. Hendrik Baier & Mark H.M. Winands. *Monte-Carlo Tree Search and Minimax Hybrids*, 2015.

7. M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu & X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

8. Chollet, F., et al. *Keras,* 2015. Software available from keras.io.