

Mate AI one

Nicola Modugno, Matteo Galiazzo, Daniele Russo,

Università di Bologna

Intelligenza Artificiale 81940

Prof. Maurizio Gabbrielli, prof. Stefano Pio Zingaro

14/04/2025

Abstract.

Questo studio esplora l'integrazione di reti neurali con un algoritmo di ricerca classico per sviluppare un motore scacchistico. Il progetto compara un approccio probabilistico che utilizza una Efficiently Updatable Neural Network (NNUE) con un algoritmo MiniMax ottimizzato.

Il lavoro valida il potenziale delle reti neurali di migliorare i motori di ricerca tradizionali, rispetto all'approccio basato su regole dell'algoritmo MiniMax, ma sottolinea la necessità di algoritmi di ricerca avanzati e ottimizzazioni hardware per colmare il divario con soluzioni SOTA come Stockfish.

Sommario

Abstract.	1
Sommario	2
1. Introduzione	3
2. Il Software	4
2.1. Struttura del sistema	5
2.2. MiniMax	5
2.3. NNUE	10
3. Dataset e preprocessing dei dati	11
4. Iperparametri e addestramento del modello	11
5. Risultati	11
Bibliografia	11

Introduzione

Un motore scacchistico è un programma per computer che analizza le posizioni degli scacchi o di una variante degli scacchi e genera una mossa o un elenco di mosse che considera più forti. La storia dell'IA negli scacchi inizia con il Turco Meccanico, costruito da Wolfgang von Kempelen nel 1769. Nonostante fosse un ingegnoso sistema meccanico manovrato da esseri umani, l'invenzione affascinò l'idea di una macchina che fosse in grado di giocare agli scacchi come un essere umano. Ciò ha spinto numerosi informatici e scienziati come Konrad Zuse, Alan Turing e David Champernowne, insieme a Claude Shannon, Dietrich Prinz e John von Neumann, allo sviluppo di programmi di scacchi sempre più sofisticati.

Gli algoritmi che simulano il comportamento umano durante una partita di scacchi si basano sulla valutazione della posizione dei pezzi sulla scacchiera per determinare le mosse migliori e prevedere l'esito della partita. Negli anni '50, Shannon propose una classificazione fondamentale: i programmi **tipo A** (brute force) analizzavano tutte le possibili mosse fino a una certa profondità, mentre i programmi **tipo B** (selettivi) cercavano di restringere il numero di mosse analizzate basandosi su criteri strategici. Tuttavia, con il tempo, l'approccio brute force si è imposto come il più efficace, soprattutto grazie agli sviluppi nell'hardware e agli algoritmi di ricerca ottimizzati.

Il nostro progetto nella parte di machine learning esplora l'utilizzo di un approccio probabilistico basato su una rete neurale artificiale basata su una **Efficiently Updatable Neural Network** (NNUE) per valutare le posizioni scacchistiche, e l'utilizzo dell'algoritmo MiniMax con potatura **alfa-beta** e beam search, per la ricerca della mossa ottima. Nella parte di intelligenza artificiale "classica" invece utilizziamo un algoritmo MiniMax fortemente

ottimizzato. L'ipotesi di partenza è che l'approccio neurale possa fornire una soluzione più scalabile ed efficiente rispetto al metodo MiniMax, sfruttando la capacità della rete di apprendere dai dati una funzione che valuta la bontà di una posizione. L'analisi si concentra sulla comparazione tra i due metodi, valutando la correttezza delle decisioni della rete neurale e la sua scalabilità, in relazione a diversi volumi di dati e configurazioni della rete.

Il Software

Un motore scacchistico è costituito da un back end, che contiene tutta la logica per il calcolo delle mosse e l'analisi della partita, e da un front end: un'interfaccia grafica con cui l'utente può interagire, muovendo i pezzi su una scacchiera virtuale.

1.1. Struttura del sistema

Il nostro motore è costituito da un front end scritto in HTML, CSS, e JavaScript. Abbiamo utilizzato le librerie chess.js, chessboardjs e jquery. Ogni volta che l'utente effettua una mossa, il front-end invia una richiesta POST al back-end per elaborare la risposta e aggiornare lo stato del gioco. Il back-end è sviluppato in Python e utilizza la libreria python-chess, che permette di mantenere una rappresentazione della scacchiera, verificare la validità delle mosse ed elaborare la logica del gioco (ad esempio, verificare condizioni di scacco, scacco matto o stallo). Nel back-end sono presenti quattro motori scacchistici: una denominata *Random*, che sceglie una mossa casuale, una denominata *AlphaBeta*, realizzata appositamente in C++ per massimizzare la velocità d'esecuzione, una basata su *Perceptron Layer* scritta con *Numpy* e infine quella chiamata NNUEP che utilizza *Pytorch*. Affinchè fosse possibile scegliere quale dei modelli utilizzare, è stata realizzata una classe chiamata *ChessEngine* che consente di selezionare una tra le reti neurali implementate. Ciascuna di queste, fornisce una funzione di valutazione che esplora i tre migliori branch ad un algoritmo MiniMax tradizionale.

Andiamo ad analizzare gli algoritmi principali e le loro ottimizzazioni ad alto livello:

1.2. MiniMax

L'algoritmo di ricerca MiniMax è semplice e riesce sempre a trovare la miglior mossa da giocare. Con gli scacchi però in media ci sono 30 mosse che un giocatore può fare in qualsiasi stato del gioco. Questo significa che ogni singolo nodo nell'albero avrà approssimativamente 30 figli diversi.

Denotiamo l'ampiezza dell'albero con w . Ci vogliono in media 85 mosse consecutive per finire una partita di scacchi. Questo significa che l'albero avrà una profondità media di 85. Denotiamo la profondità dell'albero con d .

w^d ci dice quante posizioni diverse dobbiamo valutare. Per gli scacchi il numero è quindi $30^{85} = 3.592 \times 10^{125}$. Questo ci impedisce quindi di esplorare tutte le mosse possibili.

MiniMax è basato sul fatto che un giocatore cerca di massimizzare la funzione di valutazione mentre l'altro cerca di minimizzarla. Questo comportamento si riflette direttamente nell'albero di ricerca. Infatti, durante l'attraversamento dalle foglie alla radice abbiamo sempre scelto la miglior mossa possibile per un giocatore, quindi per il bianco abbiamo scelto la miglior mossa possibile per un giocatore, e per il nero sempre il valore minimo.

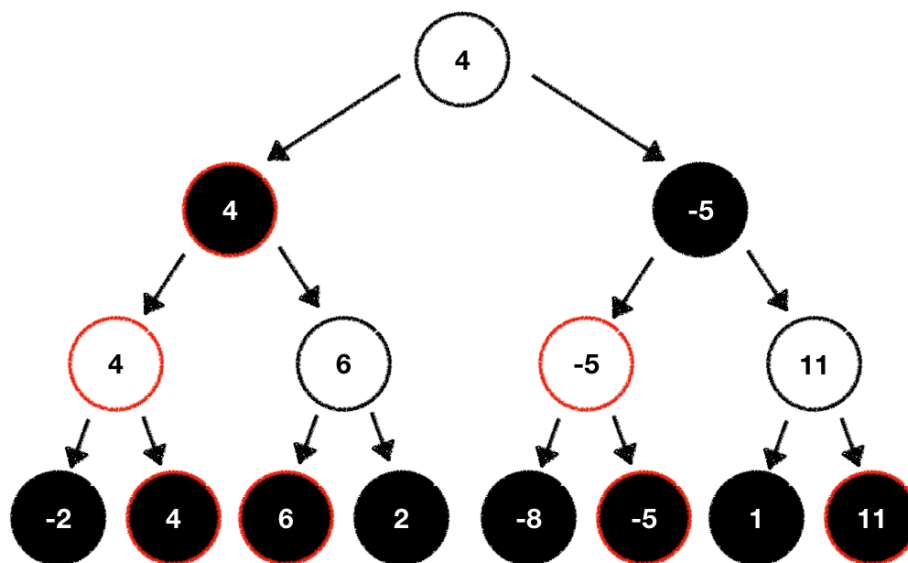


Figura 1.1. Albero di ricerca del gioco con i nodi selezionati

Se osserviamo l'albero di ricerca in **Figura 1.1** possiamo sfruttare questo comportamento per ottimizzarlo. Mentre facciamo il cammino verso le foglie dovremmo costruire il nostro albero

andando prima in profondità (depth-first). Questo vuol dire che dovremmo iniziare a un nodo ed espanderlo giocando la partita fino alla fine (fino alla profondità di ricerca desiderata) prima di tornare indietro e scegliere il nodo successivo che vogliamo esplorare. Seguire questa procedura ci permette di identificare prima le mosse che non verranno giocate. Infatti, un giocatore massimizza il numero (il risultato finale) mentre l'altro lo minimizza. La parte dell'albero di ricerca dove un giocatore finirebbe in una situazione peggiore in base alla funzione di valutazione può essere completamente rimossa dalla lista dei nodi che vogliamo espandere ed esplorare. Facciamo una potatura di questi nodi (pruning) e quindi riduciamo la larghezza dell'albero, come illustrato in **Figura 1.2**.

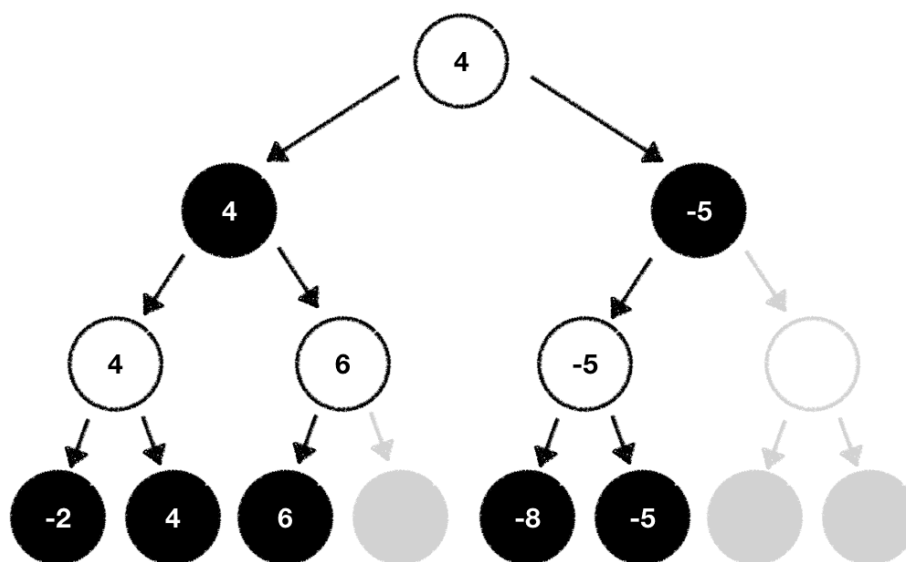


Figura 1.2. Albero di ricerca del gioco con potatura

Più alto è il branching factor dell'albero, più alto è il numero di calcoli che possiamo risparmiare.

Assumendo di poter ridurre la larghezza di una media di 10 nodi finiremmo con $w^d = (30 - 10)^{85} = 3.868 \times 10^{110}$. Questa tecnica di potatura di parti dell'albero di ricerca è chiamata alpha-beta pruning. Il nome viene dai parametri alpha e beta che vengono utilizzati

per tenere traccia del miglior punteggio che ciascun giocatore può ottenere esplorando l'albero. Il codice per l'alpha-beta pruning è illustrato di seguito in **Figura 1.3**.

```

1 eval = search_alphabeta(3, -INF, INF)
2
3 def search_alphabeta(depth, alpha, beta):
4     if depth == 0: return basic_eval()
5     moves = list(board.generate_legal_moves())
6
7     if len(moves) == 0:
8         if board.is_check(): return -INF
9         return 0
10
11    for move in moves:
12        board.push(move)
13        eval = -search_alphabeta(depth - 1, -beta, -alpha)
14        board.pop()
15        if eval >= beta: return beta
16        alpha = max(alpha, eval)
17
18    return alpha

```

Figura 1.3. Codice dell'alpha-beta pruning

Ma questa ottimizzazione non è l'unica che possiamo fare. Iniziamo aggiungendo una funzione per ordinare le mosse possibili in base a un'euristica. Cerchiamo di dare priorità alle mosse che catturano pezzi o mettono sotto scacco l'avversario. Esplorando prima queste mosse possiamo generalmente scoprire prima rami migliori dell'albero di ricerca, e fare pruning su molti più rami.

Un'altra ottimizzazione che utilizziamo sono le tabelle di trasposizione. L'ottimizzazione è quella di utilizzare la tecnica di programmazione della *memorizzazione*, che consiste nel salvare in memoria i valori restituiti da una funzione in modo da averli a disposizione per un riutilizzo successivo senza doverli ricalcolare.

Come ultima cosa abbiamo dato all'algoritmo un tempo massimo per eseguire la ricerca, di modo da poter cercare molto profondamente se la situazione lo consente (se ci sono poche mosse

possibili da esplorare) ma di mantenere comunque il programma responsivo nel caso ci siano molte mosse da cercare.

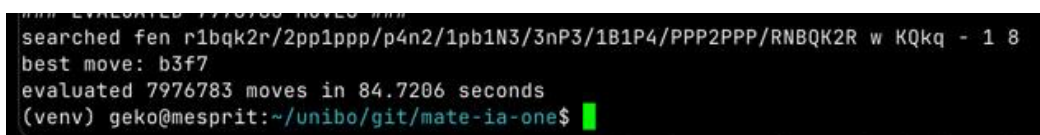
La componente di ricerca e valutazione delle mosse è stata prima prototipata in Python e poi riscritta in C++. Il passaggio a C++ è stato scelto per capitalizzare sui vantaggi in termini di performance della compilazione, fondamentali per accelerare le operazioni sequenziali.



Figura 1.4. Configurazione di una scacchiera e la sua stringa FEN

```
searched fen: r1bqk2r/2pp1ppp/p4n2/1pb1N3/3nP3/1B1P4/PPP2PPP/RNBQK2R w KQkq - 1 8
best move: b3f7
evaluated 22245796 moves in 2.22226 seconds
geko@mesprit:~/unibo/git/mate-ia-one/mate_ia_one/cpp_engine$
```

Figura 1.5. Esecuzione del codice con engine C++



```

searched fen r1bqk2r/2pp1ppp/p4n2/1pb1N3/3nP3/1B1P4/PPP2PPP/RNBQK2R w KQkq - 1 8
best move: b3f7
evaluated 7976783 moves in 84.7206 seconds
(venv) geko@mesprit:~/unibo/git/mate-ia-one$

```

Figura 1.6. Esecuzione del codice con engine Python

Come si può vedere dall'immagine, l'implementazione in C++ per la stessa posizione la ricerca della mossa alla stessa profondità impiega solo 2.2 secondi invece degli 84.7 secondi dell'algoritmo scritto in python.

1.3. NNUEP

L'implementazione con rete neurale si basa su una NNUE, un'architettura progettata per rimpiazzare la funzione di valutazione di giochi precedentemente valutati con ricerca Alpha-Beta. Dal punto di vista architetturale, la nostra rete neurale condivide con una tradizionale NNUE una struttura di tipo *feed-forward*, composta da uno strato di input che elabora vettori rappresentanti le posizioni scacchistiche (ad esempio, codifiche derivate dalla notazione FEN, con dimensione 769 (invece del tradizionale 768). Questo perché la scacchiera è 8×8 , e noi vogliamo rappresentare 2 giocatori (due colori diversi: $+1$, -1) con 6 pezzi ciascuno, ma a questa rappresentazione classica aggiungiamo anche un bit per rappresentare se è il bianco oppure il nero che deve muovere un pezzo. Quindi $8 \times 8 \times 2 \times 6 + 1 = 768 + 1 = 769$. Ciascun vettore viene poi processato da una serie di strati completamente connessi, che costituiscono la parte nascosta del modello. La rete ha 4 hidden layer, invece dei 2 layer tradizionali. In ciascun passaggio, ogni strato esegue una trasformazione lineare seguita da una funzione di attivazione. La prima importante differenza introdotta nella nostra rete neurale riguarda proprio la funzione di attivazione. Al posto della classica ReLU, viene adottata una *Clipped ReLU*, che limita i valori di attivazione in un

intervallo compreso tra 0 e 127. Questa scelta contribuisce a stabilizzare l'addestramento della rete, prevenendo la propagazione di valori troppo elevati nei layer successivi, con un impatto positivo sulla convergenza del modello. Un'altra modifica significativa è l'inclusione del dropout tra gli hidden layer. Questo meccanismo di regolarizzazione, assente nella maggior parte delle implementazioni standard di NNUE, è stato introdotto per combattere la scarsa quantità di dati che avevamo a disposizione (per via dei limiti hardware delle GPU dei nostri computer) e dato che abbiamo implementato una rete più profonda della classica NNUE. Infine, la rete neurale adotta un approccio con attention anche nella fase di inizializzazione dei pesi, utilizzando la strategia Kaiming Normal (nota anche come He Initialization). Questa tecnica è pensata appositamente per reti con attivazioni ReLU (o varianti) e assicura una migliore propagazione del gradiente nei livelli profondi, favorendo un apprendimento più stabile ed efficiente fin dalle prime epoche.

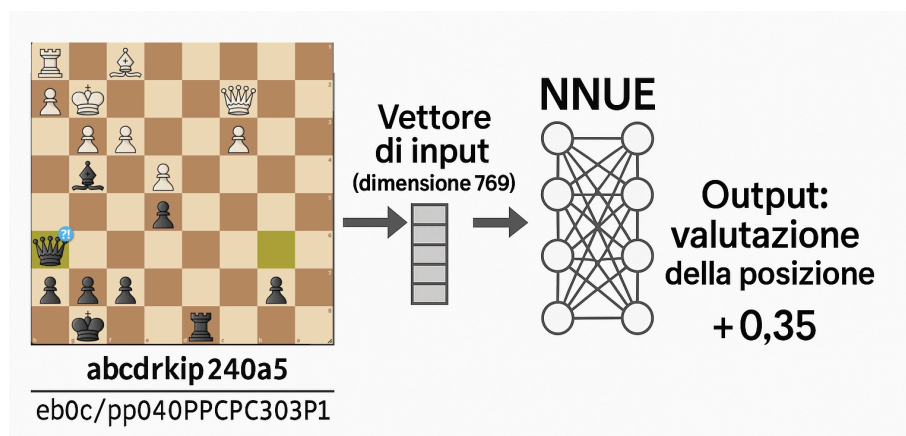


Figura 1.7. Flusso di esecuzione partendo dalla scacchiera e dalla stringa EPD

La nostra implementazione NNUEP lavora in sinergia con l'algoritmo di ricerca MiniMax con beamwidth 3. Questo significa che quando valutiamo una posizione con la NNUEP sfruttiamo la sua capacità di vedere oltre la mossa corrente (cosa che non riesce invece a fare l'algoritmo

Alpha-Beta tradizionale) per scegliere le 3 mosse che sembrano essere migliori ed espandere solo quei nodi.

In sintesi, combinando una NNUEP con una ricerca MiniMax guidata da beam search, questo approccio cerca di ottenere un motore di gioco più forte ed efficiente, dato che anche se il calcolo della funzione di valutazione tramite rete neurale è più complesso di una funzione di valutazione tradizionale, poi ci limitiamo ad espandere molti meno nodi. La rete neurale identifica le mosse a più alto potenziale, permettendo alla ricerca di concentrarsi sulle varianti più rilevanti e promettenti.

3. Dataset e preprocessing dei dati

Prima di addestrare la rete neurale sono stati individuati diversi dataset per l'addestramento di motori scacchistici, tra questi, sono stati ritenuti ottimali, per il nostro caso d'uso e risorse a disposizione, solo alcuni che sono stati selezionati per la fase di training, validation e test. Per la fase di training è stato selezionato il dataset *Chess Evaluation* disponibile su [Kaggle](#). Dato che abbiamo utilizzato un subset del dataset originale (che contiene 16 milioni di posizioni), addestrare il motore scacchistico utilizzando il dataset "così com'è" avrebbe potuto comportare il rischio che il modello venisse addestrato su sequenze non rappresentative dell'intera distribuzione, compromettendo la sua capacità di generalizzare su nuovi dati. Per questo motivo abbiamo mescolato le righe del dataset.

4. Iperparametri e addestramento del modello

L'obiettivo principale del lavoro in questione è quello di confrontare l'approccio probabilistico rispetto all'approccio deterministico (rule based). Le reti neurali sono state valutate prendendo in

considerazione la variazione delle performance rispetto allo scalare delle sue dimensioni (come il numero di nodi per hidden layer) e della quantità di dati per ogni addestramento, in particolare, il numero di righe prese in considerazione nel dataset. Inoltre, l'addestramento è stato eseguito su un numero variabile di epoche e mantenendo lo stesso dropout per ogni layer. Per la fase di test è stato utilizzato l' **Eigenmann Rapid Engine Test** (ERET): una collezione di 111 posizioni di test che coprono un ampio spettro di motivi e temi scacchistici, disponibile su [Chessprogramming wiki](https://chessprogramming.wiki). Ciascun dataset è strutturato in una sequenza di righe, ciascuna contenente una stringa EPD e la sua corrispettiva valutazione, ad eccezione del file dell'ERET che al posto della valutazione dello stato della scacchiera contiene la mossa ottima da eseguire.

Abbiamo valutato attentamente gli iperparametri su cui potevamo lavorare, e dopo aver confermato le nostre idee tramite degli addestramenti della rete su piccoli subset dei dati abbiamo scelto di:

- Fissare l'input size a 769, per le considerazioni sulla rappresentazione della stringa FEN nella NNUEP spiegate nel capitolo 1.3.
- Fissare un massimo di 100 epoche, che viene interrotto precocemente se la validation loss aumenta eccessivamente.
- Fissare un massimo di 300 k posizioni scacchistiche nel nostro dataset, per limiti sulla memoria RAM dei computer su cui abbiamo addestrato la rete.
- Dropout variabile tra 0.1 e 0.3. È stato osservato che un valore di dropout intorno a 0.1 ha migliorato il punteggio R^2 , passato da una media iniziale di 0.175, fino a un massimo di 0.1870.
- Testare diverse ipotesi sugli hidden layer:

Il processo di scelta degli iperparametri è stato guidato da un processo iterativo di *trial and error*, con l'obiettivo di massimizzare le prestazioni della rete e l'efficienza computazionale. La rete è stata addestrata su dataset di diversa grandezza. In una prima iterazione sono state utilizzate 1.500.000 righe del dataset provenienti da tre differenti file per l'addestramento. In una fase successiva, è stato anche utilizzato un dataset ridotto a 300.000 istanze per verificare l'impatto dell'ampiezza del training set sull'accuratezza. L'input size è stato fissato a 769, mentre il numero di epoche è stato mantenuto al di sotto di 100 per contenere i tempi di calcolo, garantendo al contempo una buona convergenza. Il parametro di dropout, variato tra 0.1 e 0.3, ha permesso di controllare l'overfitting. È stato osservato che un valore di dropout intorno a 0.1 ha migliorato il punteggio R^2 , passato da una media iniziale di 0.175, che con un dropout medio di 0.125 è stato ridotto fino a un massimo di 0.1870. Il numero di neuroni nei layer nascosti (hidden size) è stato sperimentato con diverse configurazioni per testare il modello, diminuendone la dimensione fino a trovare quella più adeguata per l'addestramento con 300.000 righe di test. Ognuna di queste configurazioni sono state confrontate tra loro. La K-Fold Cross Validation: una tecnica che consiste nella suddivisione del training set in parti uguali di cui una costituisce il validation set e le rimanenti compongono il training dataset ci ha consentito di confrontare diverse porzioni dell'insieme dei dati, così da distinguere quali tra questi fosse di scarsa qualità eliminandoli dall'insieme per l'addestramento. Il dataset è stato suddiviso in 5 e, in una successiva iterazione, in 2 parti e, per ognuna di esse, l'80% ha costituito il training set ed il 20% il validation set. L'addestramento della rete neurale è stato ottimizzato attraverso una serie di esperimenti sistematici guidati dai risultati, che verranno illustrati di seguito:

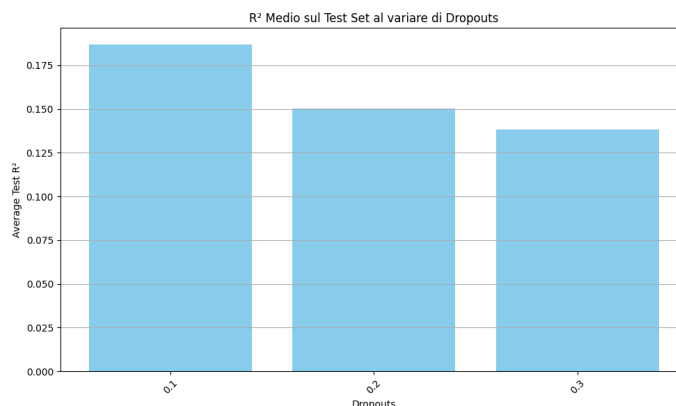


Figura 4.0 Grafico comparazione dei valori dropouts

Dropout: Variando il tasso di dropout tra 0.1 e 0.3, il valore ottimale è risultato 0.1, con un R^2 medio sul test set di 0.187 (Figura 4.0). Valori superiori a 0.2 hanno ridotto drasticamente le prestazioni, indicando un eccessivo underfitting.

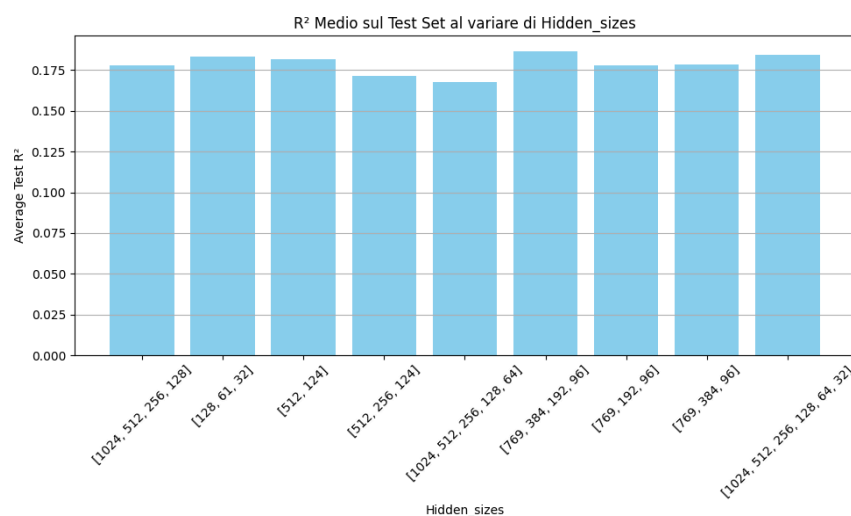


Figura 4.1 Grafico comparazione degli hidden layer e hidden size

Dimensioni dei layer nascosti: Le configurazioni con architetture più profonde (es. [1024, 512, 256, 128]) hanno ottenuto un R^2 medio di 0.175, superando reti più piccole come [128, 81, 32] (Figura 4.1). Ciò conferma che modelli più complessi catturano meglio le relazioni nelle posizioni scacchistiche.

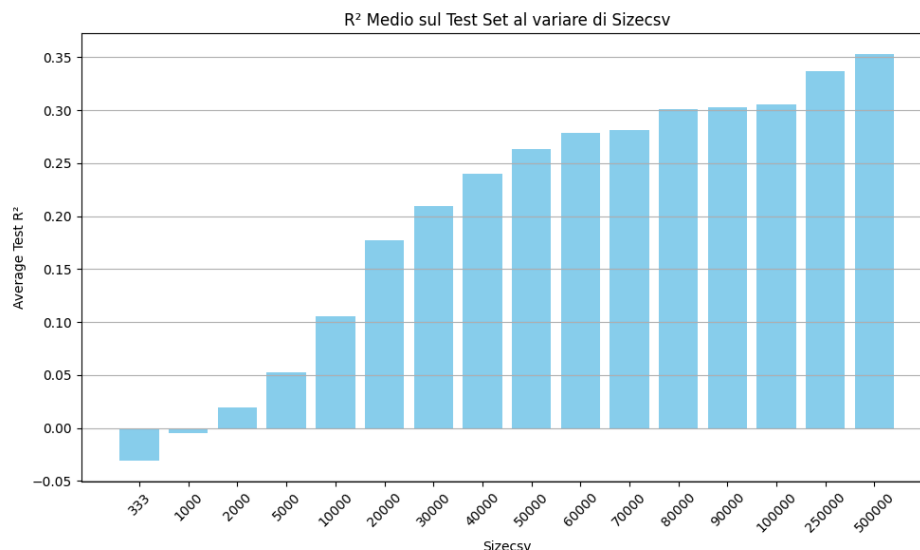


Figura 4.2 Grafico comparazione degli hidden layer e hidden size

Scalabilità con la dimensione del dataset: Utilizzando dataset da 1k a 1.5M istanze (il numero riportato nel grafico sarebbe in numero di istanze per file analizzato, nello specifico 3 file diverse, questo porta a dover moltiplicare il numero indicato per tre), l' R^2 è migliorato da -0.05 a 0.35, dimostrando una chiara correlazione tra volume dei dati e prestazioni (Figura 4.2).

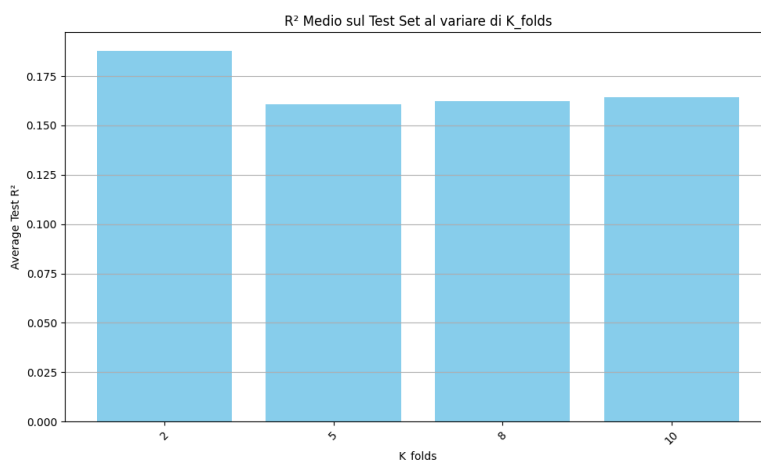


Figura 4.3 Grafico comparazione dei k-fold

K-Fold Cross Validation: Suddividendo il dataset in 5 fold, la varianza tra le fold è risultata minima (± 0.02), indicando una distribuzione equilibrata dei dati (Figura 4.3).

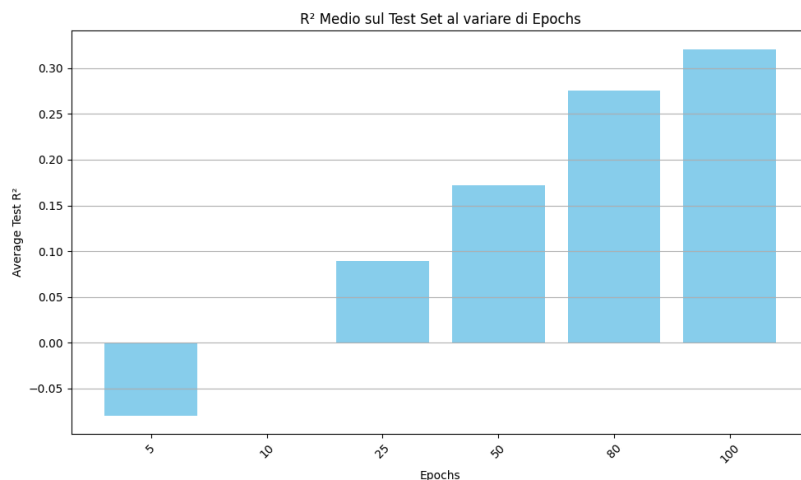


Figura 4.4 Grafico comparazione degli numero di epoche

Epoche: Vista la grande quantità di esempi e la complessità del task l' R^2 ha mostrato anche una significativa correlazione con le epoche (Figura 4.4).

I risultati migliori si sono ottenuti con in input size di 769, quattro hidden layer con valori [769,384,192,96] un dropout di 0.1 con una size di 1.5 M di esempi su 100 epoche.

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	AVG
MSE	0.1788	0.1854	0.1874	0.1778	0.1877	0.1834
RMSE	0.4229	0.4305	0.4329	0.4217	0.4332	0.4283
MAE	0.3210	0.3280	0.3307	0.3196	0.3316	0.3262
R^2	0.3672	0.3466	0.3395	0.3753	0.3375	0.3532

Tabella 1.1. Risultati dei migliori iperparametri

5. Risultati

5.1. Selfplay

Per avere un confronto diretto tra i vari algoritmi abbiamo fatto giocare tra di loro i due engine.

L'algoritmo NNUEP ha giocato potendo esplorare le mosse fino a profondità 4 e con i migliori iperparametri che abbiamo individuato precedentemente (InputSize: 769, HiddenSizes:[769, 384, 192, 96], dropout:0.1, Dataset Size:1,5 M).

L'algoritmo MiniMax ha giocato senza un massimo di profondità di ricerca, ma dovendo effettuare la mossa dopo 3 secondi. Delle 10 partite che sono state giocate, NNUEP ne ha vinte 4, e ci sono stati 6 pareggi.

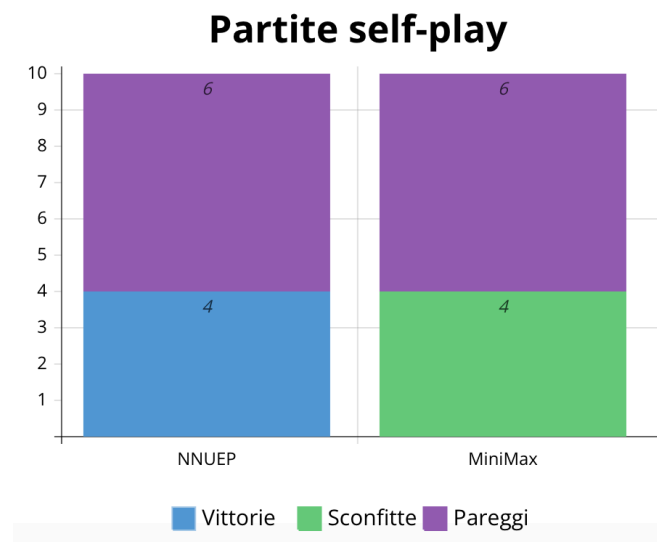


Figura 5.1 Grafico comparazione degli hidden layer e hidden size

L'algoritmo MiniMax non è quindi mai riuscito a vincere una partita (Figura 5.1). Abbiamo notato che i 6 pareggi sono avvenuti:

- Per five fold repetition 1 volta.
- Per materiale insufficiente 1 volta.
- Per stalemate 1 volta.

- Per regola delle 75 mosse 3 volte.

La regola delle 75 mosse dice che se non avviene una cattura o mossa di pedone nelle ultime 75 mosse la partita è automaticamente pareggiata. Il fatto che questa situazione capiti 3 volte in sole 10 partite ci fa intuire che entrambi gli algoritmi faticano a trovare una mossa che conclude la partita (o che cattura un pezzo dell'avversario), seppur nell'algoritmo MiniMax sia anche implementato il codice per dare priorità alle mosse che danno scacco o catturano pezzi.

5.2. ERET

Abbiamo anche deciso, come già accennato, di testare i modelli tramite ERET. Il test ERET consiste in un insieme di posizioni scacchistiche per cui è difficile trovare la migliore mossa da eseguire.

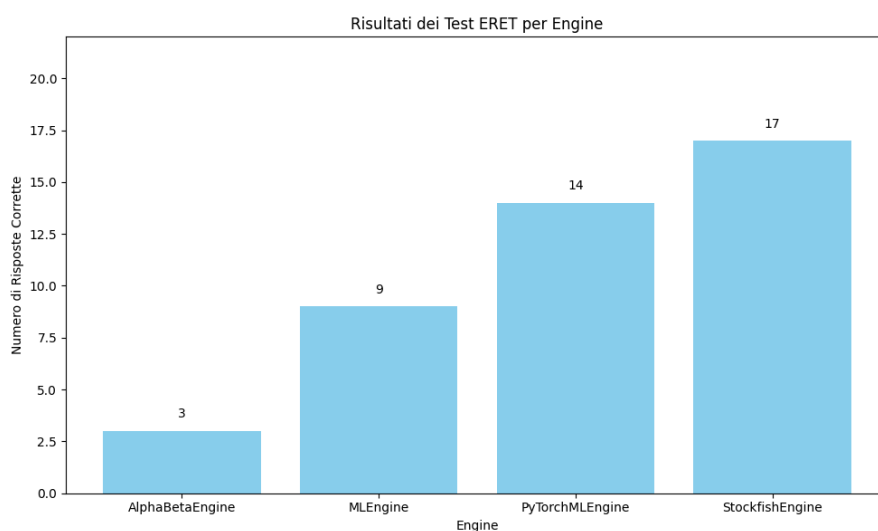


Figura 5.2.1 Test ERET a profondità 4

I test su ERET hanno dato i seguenti risultati:

- AlphaBetaEngine (MiniMax): Solo 3/111 risposte corrette a profondità 4, a causa del branching factor elevato e della limitata profondità di ricerca.

- NNUE con 2 layer: 9/111 risposte corrette, sfruttando la valutazione probabilistica delle posizioni.
- NNUE con 4 layer (nostra implementazione): 14/111 risposte corrette, con un miglioramento del 55% rispetto alla versione a 2 layer.
- Stockfish: 17/111 risposte corrette

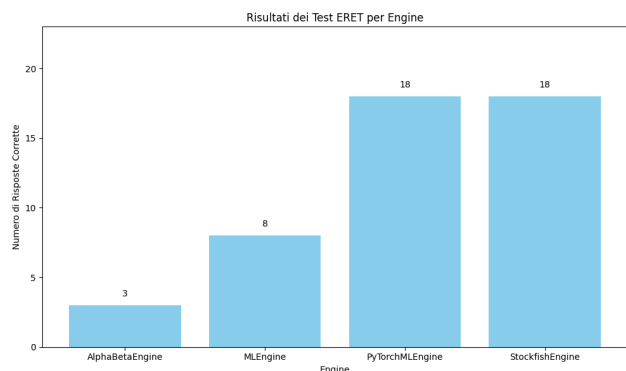


Figura 5.2.3. Test a profondità 2

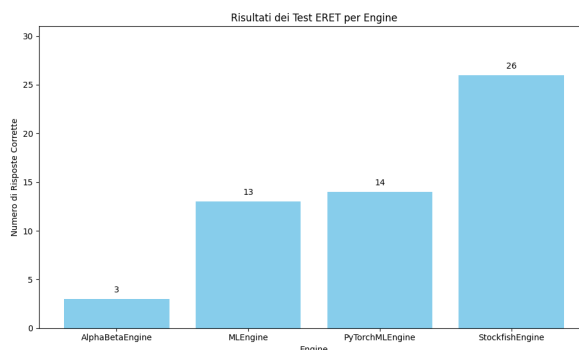


Figura 5.2.3. Test a profondità 6

L'approccio data-driven con NNUEP ha dimostrato:

1. Scalabilità: Le prestazioni migliorano linearmente con la dimensione del dataset e la complessità del modello.
2. Efficienza: La valutazione probabilistica riduce significativamente il numero di nodi esplorati rispetto al MiniMax, ottimizzando i tempi di calcolo.
3. Performance: La NNUEP a 4 layer predice correttamente 4 volte più posizioni dell'algoritmo AlphaBeta nell'ERET, ma rimane peggiore di Stockfish, probabilmente per le migliori ottimizzazioni fatte nel programma (ricerca multicore, tabelle di trasposizione, ripartenza dal controllo della migliore mossa precedente...).

Questi risultati supportano l'ipotesi iniziale: le reti neurali, combinate con tecniche di ricerca ottimizzate, offrono un vantaggio significativo rispetto ai metodi rule-based tradizionali.

Tuttavia, l'integrazione con algoritmi di ricerca più avanzati (es. Monte Carlo Tree Search) e l'utilizzo di hardware più performante e dedicato potrebbe colmare il gap con motori all'avanguardia.

Bibliografia

- [1] V. Vuckovic, "Candidate Moves Method implementation in MiniMax search procedure of the Achilles chess engine," 2015 12th International Conference on Telecommunication in Modern Satellite, Cable and Broadcasting Services (TELSIKS), Nis, Serbia, 2015, pp. 314-317, doi: 10.1109/TELSIKS.2015.7357795
- [2] Sreedhar, Suhas (2 July 2007). "Checkers, Solved!". IEEE Spectrum. Institute of Electrical and Electronics Engineers
- [3] "Stockfish 12". [Stockfish Blog](#)
- [4] "Chessprogramming wiki". www.chessprogramming.org
- [5] "Stockfish FAQ: Can Stockfish use my GPU?". Stockfish
- [6] [nnue-pytorch/docs/nnue.md](https://nnue.pytorch/docs/nnue.md)
- [7] Dominik Klein, [Neural Networks for Chess](#), p. 49
- [8] Monroe, Daniel; Chalmers, Philip A. (2024-10-28), [Mastering Chess with a Transformer Model](#)