

Funktionsweise der Schach-KI Stockfish und Implementation einer eigener Schach-KI mittels eines CNN basierend auf Supervised-Learning

Studienprojekt

im Studiengang
Softwaretechnik und Medieninformatik
der Fakultät Informationstechnik

vorgelegt von
Mateusz Frydryszak
Matrikelnummer: 767398

am 19. Juni 2025
an der Hochschule Esslingen

Erstprüfer: Prof. Dr. Ing. Thao Dang
Zweitprüfer: Prof. Dr. rer. nat. Markus Enzweiler
Zeitraum: 01.05.2025 - 31.08.2025

Ehrenwörtliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 19. Juni 2025

Unterschrift

Gender Erklärung

Das in dieser Arbeit gewählte generische Maskulinum bezieht sich zugleich auf die männliche, die weibliche und andere Geschlechteridentitäten. Zur besseren Lesbarkeit wird auf die Verwendung männlicher und weiblicher Sprachformen verzichtet. Alle Geschlechteridentitäten werden ausdrücklich mitgemeint, soweit die Aussagen dies erfordern.

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Tabellenverzeichnis	II
Quellcodeverzeichnis	III
1 Einleitung	1
2 Schach Repräsentierung	2
2.1 Bitboard-Repräsentation	2
2.2 Spielfiguren-Unterscheidung	3
2.2.1 Springende Figuren (Bauer, Springer, König)	3
2.2.2 Gleitende Figuren (Läufer, Turm, Dame)	4
2.3 Magic-Bitboards	5
2.4 Hash-Verfahren	6
2.5 Zug-Generierung	7
3 Funktionsweise von Stockfish	8
3.1 Mini-Max Algorithmus	8
3.2 Alpha-Beta Pruning	9
3.2.1 Pruning-Beispiel	10
4 Evaluationsfunktion	11
5 Schach KI Implementierung	13
5.1 Ansatz mit Convolutional Neural Networks	13
5.2 Problembeschreibung	13
5.3 Datensatz	13
5.4 Schachbrett Repräsentierung	13
5.5 Zug-Index Konversion	15
5.5.1 UCI-Format	16
5.6 Architektur	17
5.7 Evaluation	17
6 Fazit	18
Literaturverzeichnis	19

Abbildungsverzeichnis

2.1	Schachbrett-Stellung	2
2.2	Mögliche Felder für den Springer	4
2.3	Der a1-Turm wird auf a5 und f1 in seiner geraden Bewegungslinie blockiert	5
3.1	Minimax Suchbaum mit Tiefe 4	8
3.2	Alpha-Beta Suchbaum mit eliminierten Zweigen	10
5.1	Forward-Pass des Basic Modells	17

Tabellenverzeichnis

2.1	Bitboard-Darstellung der Bauernposition	3
2.2	Bitmaske der relevanten Felder für den Turm	6
3.1	Suchbaum-Wachstum und Suchzeiten	9

Quellcodeverzeichnis

2.1	Springer Zug Felder Berechnung	3
2.2	Hash-Verfahren mit Magic Bitboards	7
5.1	Funktion zur Encodierung eines Schachbretts	15

1 Einleitung

Die Herausforderung, einen Computer zum Schachspielen zu befähigen, beschäftigte bereits Alan Turing und seine Studierenden [12]. Was das Schachspiel besonders macht, ist die enorme Vielfalt möglicher Spielverläufe. Schätzungen zufolge gibt es nach 40 Zügen etwa 10^{120} verschiedene mögliche Stellungen auf dem Brett – eine Zahl, die sogar die geschätzte Anzahl der Atome im beobachtbaren Universum (etwa 10^{80}) übertrifft [5]. Die Berechnung optimaler Schachzüge ist eine immense Herausforderung, die selbst modernste Computer an ihre Grenzen bringt. Eine vollständige Durchsuchung aller möglichen Spielverläufe wäre mit heutiger Rechenleistung nicht realisierbar. Ein einfacher Brute-Force-Ansatz ist daher praktisch unmöglich. Daher widmet sich dieses Studienprojekt der Analyse, wie moderne Schach-KIs, insbesondere Stockfish[9], diese Problematik bewältigen und welche faszinierenden Algorithmen und Heuristiken sie dabei verwenden. Es wird ebenfalls eine Architektur einer einfachen Schach-KI vorgestellt und umgesetzt, um einen näheren Einblick in die Funktionsweise und Programmierung einer Schach-Engine zu geben.

2 Schach Repräsentierung

Stockfish zählt zu den bekanntesten und leistungsfähigsten Schach-Engines weltweit. Als Open-Source-Programm wird es kontinuierlich von einer aktiven Community weiterentwickelt. Es verfügt über die typischen Komponenten einer Schach-Engine wie Suchalgorithmus, Bewertungsfunktion, Bitboard-Repräsentation und heuristische Verfahren.

2.1 Bitboard-Repräsentation

Um ein Schachbrett digital abbilden zu können, wird eine geeignete Datenstruktur benötigt, die sowohl kompakt als auch leistungsfähig bei der Verarbeitung ist. Stockfish verwendet hierfür sogenannte *Bitboards*: 64-Bit-Ganzzahlen, bei denen jedes Bit einem bestimmten Feld des 8×8 -Bretts entspricht. Diese Darstellung ermöglicht extrem schnelle bitweise Operationen (AND, OR, XOR etc.), die die Grundlage der Evaluationsfunktion bilden.

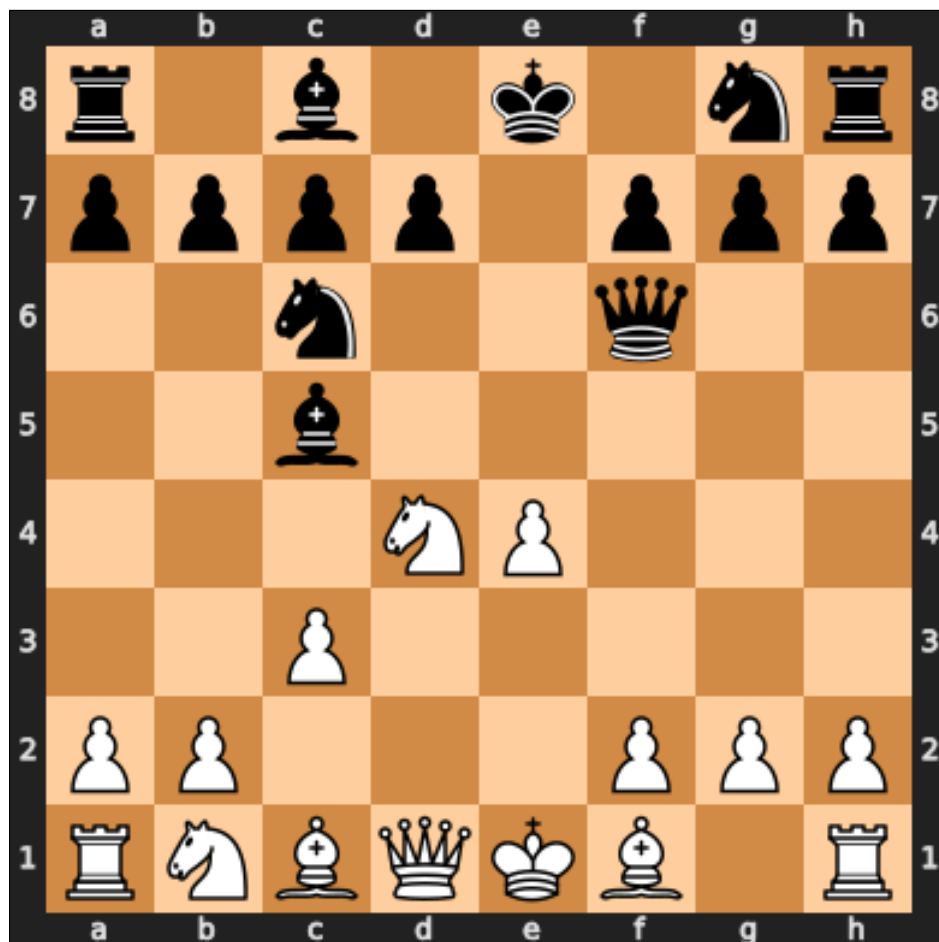


Abbildung 2.1: Schachbrett-Stellung *Quelle:[4]*

Beispiel:

Die Position der Bauern in Abbildung 2.1 lässt sich durch ein Bitboard folgendermaßen darstellen:

.
1	1	1	1	.	1	1	1
.
.
.	.	.	.	1	.	.	.
.	.	1
1	1	.	.	.	1	1	1
.

Tabelle 2.1: Bitboard-Darstellung der Bauernposition

2.2 Spielfiguren-Unterscheidung

Für die Schachzug-Generierung unterscheidet man zwischen sogenannten *springenden Figuren* (englisch: *Leaping Pieces*) und *gleitenden Figuren* (englisch: *Sliding Pieces*). Diese Unterscheidung basiert darauf, dass sich diese beiden Figurentypen in ihren Bewegungsmöglichkeiten und Einschränkungen deutlich unterscheiden.

2.2.1 Springende Figuren (Bauer, Springer, König)

Die Zugmöglichkeiten der springenden Figuren sind im Voraus berechnet und in einer Lookup-Tabelle hinterlegt. Während des Spiels kann man über den Index der aktuellen Position der Figur direkt auf diese Tabelle zugreifen und erhält als Ergebnis ein Bitboard, auf dem alle erlaubten Zielfelder mit einer „1“ markiert sind. Durch diese vorher erstellte Tabelle wird viel Zeit gespart, da die Züge nicht berechnet, sondern einfach nachgeschlagen werden.

```
1 //leeres Spielfeld
2 U64 attacks, knights = 0ULL;
3
4 // platziert Springer auf dem Brett
5 set_bit(knights, square);
6
7 // Springer Zuege lassen sich mithilfe von Bit-Shifts generieren
8 attacks = (((knights >> 6) | (knights << 10)) & ~FILE_GH) |
9           (((knights >> 10) | (knights << 6)) & ~FILE_AB) |
10          (((knights >> 15) | (knights << 17)) & ~FILE_H) |
11          (((knights >> 17) | (knights << 15)) & ~FILE_A);
```

Quellcode 2.1: Springer Zug Felder Berechnung

Erklärung zur Berechnung: Der Springer bewegt sich in Form eines „L“. Ein Bit-Shift »17 entspricht dabei einer Verschiebung um zwei Reihen (16 Bit) nach oben sowie einem Feld (1 Bit) nach rechts. Analog steht ein Bit-Shift «10 für eine Verschiebung um eine Reihe (8 Bit) nach oben und zwei Felder (2 Bit) nach rechts [1].

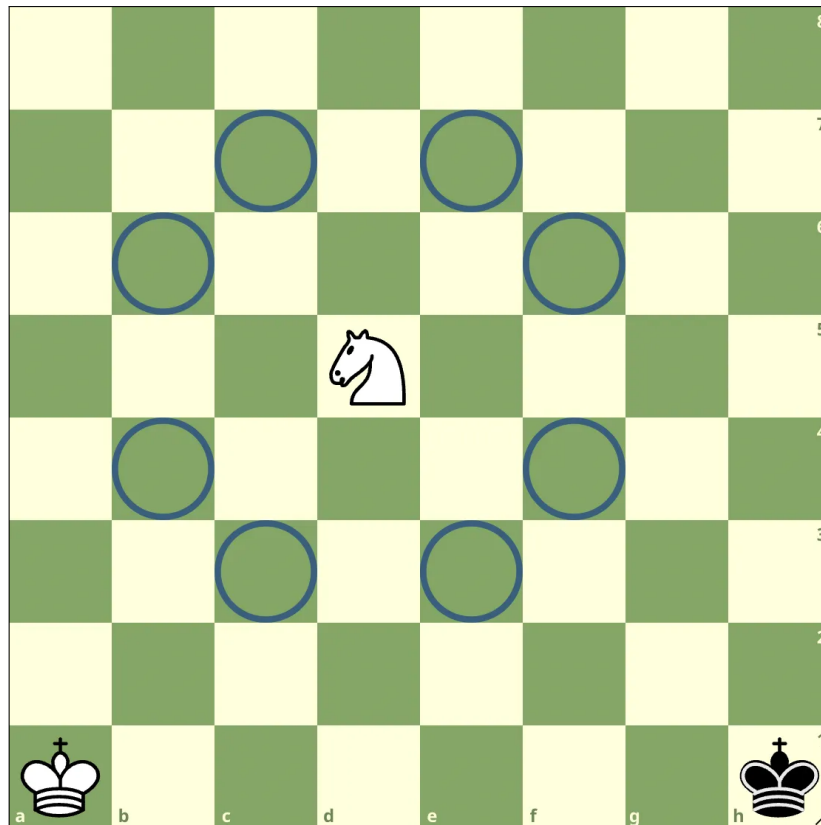


Abbildung 2.2: Mögliche Felder für den Springer *Quelle:[1]*

2.2.2 Gleitende Figuren (Läufer, Turm, Dame)

Gleitende Figuren wie Läufer, Türme und Damen können mehrere Felder in gerader Linie ziehen, solange sie keinem *Blocker* (eigene oder gegnerische Figur) begegnen.

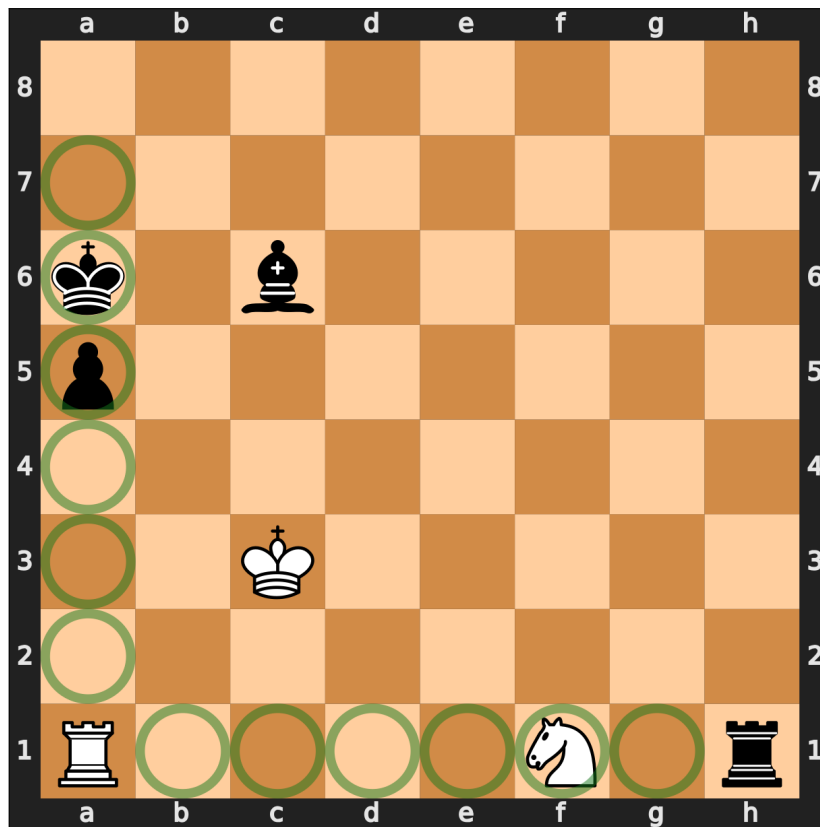


Abbildung 2.3: Der a1-Turm wird auf a5 und f1 in seiner geraden Bewegungslinie blockiert
Quelle:[4]

Das heißt, wenn man die möglichen Züge der Gleitfiguren dynamisch berechnen wollte, müsste man folgende Schritte durchlaufen:

1. Feld für Feld in jede mögliche Richtung iterieren (im Worst Case bis zu 6 Felder weit),
2. Jedes dieser Felder daraufhin prüfen, ob es besetzt ist,
3. Die Iteration beenden, sobald ein Blocker erkannt wird,
4. Wenn der Blocker eine gegnerische Figur ist, darf das Feld mit einbezogen werden – bei einer eigenen Figur nicht.

Da dieser Vorgang bei Türmen und Läufern jeweils in vier Richtungen durchgeführt werden muss (bei der Dame sogar in acht), summiert sich der Rechenaufwand schnell auf. Für eine leistungsfähige Schach-KI ist das auf Dauer zu ineffizient. Um dieses Problem zu umgehen, nutzt man sogenannte **Magic Bitboards** – eine Technik, mit der man die möglichen Züge der Gleitfiguren blitzschnell per Lookup berechnen nachschlagen kann.

2.3 Magic-Bitboards

Ansatz: Alle möglichen Blocker-Konfigurationen im Voraus berechnen und daraus einen Index erzeugen, der als Schlüssel für eine Nachschlagetabelle dient. Diese Tabelle enthält zu

jedem Schlüssel die zugehörigen möglichen Zugfelder. Die relevanten Felder für den a1-Turm in Abbildung 2.3 lassen sich als folgende Bitmaske kodieren:

.
1
1
1
1
1
1
.	1	1	1	1	1	1	.

Tabelle 2.2: Bitmaske der relevanten Felder für den Turm

Man beachte, dass die Felder a8 und h1 für die Betrachtung nicht relevant sind, denn sie können keinen weiteren Felder als Blocker dienen.

Das Ziel ist daher, eine Funktion zu finden, die diese $2^{12} = 4096$ distinkten Blockerkonfigurationen ihrer jeweiligen Menge an möglichen Zügen zuordnet. Die Reduzierung von 2^{64} (das gesamte Schachbrett) auf 2^{12} Konfigurationen ist bereits erfreulich, bringt jedoch das Problem **zerstreuter Indizes** mit sich. Eine ideale Nachschlagetabelle sähe so aus:

```
1 table[0], table[1], ... , table[4095]
```

- Die Indizes sind konsekutiv und lückenlos.
- Sie liegen ausschließlich im Bereich 0 bis $2^{12} - 1$.

In der Praxis enthält der 64-Bit-Wert jedoch nur 12 relevante Bits, die über den gesamten 64-Bit-Raum verteilt sein können. Ein Zugriff könnte zum Beispiel so aussehen:

```
1 0x00000000000000010
2 0x00000000100000000
3 0x00010000000000000
```

Obwohl jede Blockerkonfiguration nur 12 Bits nutzt, beansprucht sie den gesamten 64-Bit-Adressraum. Das macht deutlich, warum hier eine **Hashfunktion** benötigt wird, die diesen Raum auf kompakte, aufeinanderfolgende Indizes abbildet und damit ein einfaches Lookup ermöglicht.

2.4 Hash-Verfahren

Im Folgenden wird das Hash-Verfahren zur Indexberechnung bei Magic Bitboards dargestellt. Die Kommentare im Code erklären die einzelnen Schritte:

```
1  uint64_t blockers = // 64-Bit kodierte Schachbrett, wobei nur die tiefsten 12 Bits
    gesetzt sind
2  uint64_t magic     = // vorgerechnete "magische" Zahl
3
4                      // die Multiplikation bewirkt ein "Durchmischen" der gesetzten
                        Bits
5  int index          = (blockers * magic) >> (64 - 12);
6  //der abschließende Shift liefert die 12 höchsten Bits
7  // Index ist somit eine Zahl zwischen 0 und 4095
```

Quellcode 2.2: Hash-Verfahren mit Magic Bitboards

Die *magische* Konstante wird nicht willkürlich gewählt, sondern im Rahmen eines gezielten Trial-and-Error-Verfahrens für jede Figur und jedes Ausgangsfeld ermittelt. Hierfür werden zufällig ausgewählte 64-Bit-Kandidaten generiert, die bestimmte Eigenschaften erfüllen (z. B. eine geringe Bitdichte in den höheren Positionen). Liefert eine solche *Magic*-Zahl für alle 2^{12} Blocker-Muster eindeutige Indizes ohne Kollision, gilt sie als valide *magische* Konstante und wird gespeichert sowie für alle weiteren Berechnungen wiederverwendet. Das Verfahren gewährleistet eine kollisionsfreie Lookup-Tabelle, die jeder Blocker konfiguration einen eindeutigen Index zuweist und so eine schnelle Zuordnung zu den zugehörigen Zugfeldern ermöglicht.

2.5 Zug-Generierung

Stockfish implementiert die Schachzug-Generierung in einem zweistufigen Verfahren, das zunächst pseudolegale Züge (d. h. inklusive Züge, die den Schachregeln nach unzulässig sind) erzeugt und anschließend diese Menge auf legale Züge reduziert. Diese Trennung ist sinnvoll, weil:

- das Erzeugen pseudolegaler Züge extrem schnell über Bitoperationen möglich ist,
- eine vollständige Legalitätsprüfung für alle Züge ineffizient wäre,
- in vielen Spielsituationen nur wenige Züge tatsächlich illegal sind.

Die Zuggenerierung für springende Figuren erfolgt vollständig über Lookup-Tabellen, in denen für jedes Ausgangsfeld eine Attacke-Maske hinterlegt ist (siehe Kapitel 2.2.1). Für gleitende Figuren kommen hingegen die zuvor beschriebenen Magic Bitboards zum Einsatz: Nach einigen effizienten Bit-Operationen wird ein Index berechnet, der direkt auf eine vorab erstellte Attacke-Tabelle verweist.

3 Funktionsweise von Stockfish

3.1 Mini-Max Algorithmus

Die Spieltheorie bezeichnet Spiele, bei denen die Summen der Gewinne und Verluste aller Spieler gleich null sind, als *Nullsummenspiele*. Mini-Max ist einer der gängigen Algorithmen, der bei Nullsummenspielen wie Schach oder Tic-Tac-Toe eingesetzt wird. Stockfish verwendet Mini-Max, um mögliche Züge als Knoten in einem Suchbaum darzustellen und jedem Knoten mithilfe einer Bewertungsfunktion einen numerischen Wert zuzuweisen, der die Qualität des Zuges repräsentiert. Das Besondere am Mini-Max-Algorithmus ist, dass er nicht nur den aktuell besten eigenen Zug sucht, sondern auch berücksichtigt, welcher Zug den Gegner in seinen zukünftigen Optionen am stärksten einschränkt. Dabei übernimmt ein Spieler die Rolle des *Maximierers*, der darauf abzielt, den höchstmöglichen Bewertungswert zu erreichen. Der Gegenspieler agiert als *Minimierer* und versucht, den niedrigstmöglichen Wert zu erzwingen – also die für den Maximierer ungünstigste Fortsetzung. Auf diese Weise wird nicht nur der eigene Vorteil maximiert, sondern zugleich der Handlungsspielraum des Gegners minimiert. In der Schachbewertung gilt dabei das Vorzeichenkonventionsschema:

$v > 0$ bedeutet Vorteil für Weiß, $v < 0$ bedeutet Vorteil für Schwarz.

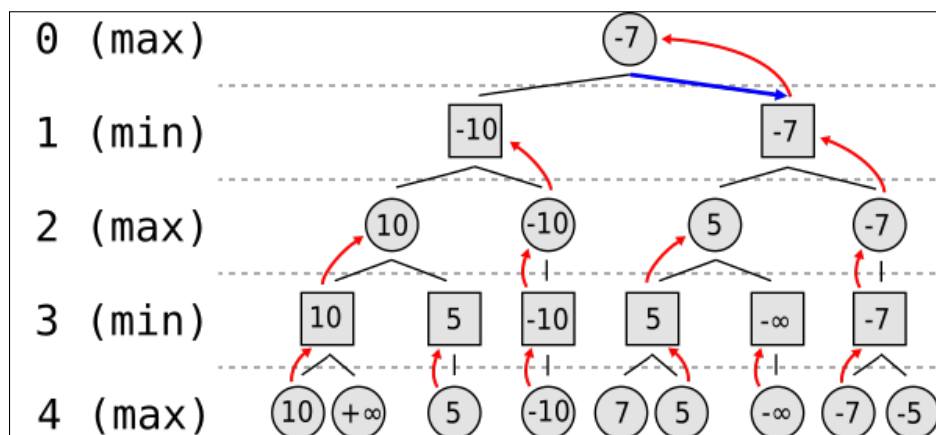


Abbildung 3.1: Minimax Suchbaum mit Tiefe 4 *Quelle:[13]*

Minimax erzeugt den Suchbaum, indem es eine festgelegte Tiefe vorausschaut. Mithilfe von **Alpha-Beta-Pruning** werden Zweige eliminiert, die voraussichtlich keine vielversprechenden Stellungen liefern können. Die obige Abbildung zeigt einen Beispielbaum, der die Berechnung des optimalen Zuges veranschaulicht. Weiß (runde Knoten) ist am Zug und strebt danach, den Wert zu maximieren. In Tiefe 4 ist ein Zug mit dem Wert $+\infty$ sichtbar, der Weiß wahrscheinlich den Sieg sichern würde. Allerdings blockiert Schwarz (eckige Knoten) diesen Zug als Minimierer und wählt stattdessen Knoten 10. Das Blatt mit dem Wert 10 wird daher hochgeschoben bzw. „rückpropagiert“. Dieser Prozess wird rekursiv für alle Knoten

wiederholt, bis die Werte an die Wurzel des Baums zurückgegeben werden. Dort trifft der Spieler die Entscheidung, sich für den minimalen oder, wie in diesem Fall, den maximalen Wertknoten zu entscheiden. Weiß wählt somit den Zug mit der Bewertung -7 , da er dort für sich das lokale Maximum sieht. Ein großer Nachteil bei der vollständigen Analyse **aller** möglichen Spielzüge ist der rapide ansteigende Rechenaufwand, da der Suchraum mit jedem Zug enorm zunimmt. Geht man beispielsweise von durchschnittlich $b = 30$ möglichen Zügen pro Stellung aus und nimmt an, dass ein Programm $R = 50\,000$ Stellungen pro Sekunde berechnen kann, ergeben sich folgende beispielhafte Suchzeiten:

Tiefe (ply)	Anzahl Stellungen	Suchzeit
2	900	0,018 s
3	27 000	0,54 s
4	810 000	16,2 s
5	24 300 000	8 Minuten
6	729 000 000	4 Stunden
7	21 870 000 000	5 Tage

Tabelle 3.1: Suchbaum-Wachstum und Suchzeiten

Es ist ersichtlich, dass der Suchbaum schnell extrem groß wird trotz einer schnellen Bewertungsfunktion. Eine zuverlässige Suche der Tiefe 6 bis 7 ist jedoch Voraussetzung für eine gute Schach-KI. Dies verdeutlicht die Notwendigkeit, die Größe des Suchbaums erheblich zu reduzieren. Hierfür kommt das Alpha-Beta-Suchverfahren zum Einsatz.

3.2 Alpha–Beta Pruning

Alpha–Beta Pruning ist ein Algorithmus, der dazu dient, Zweige eines Suchbaums zu ignorieren, die nicht zu einem vorteilhaften Zug führen. Dabei werden die Parameter

$$\alpha = -\infty, \quad \beta = +\infty$$

initialisiert. Im Verlauf der rekursiven Suche repräsentiert

- α den aktuell *besten* (*höchsten*) Wert, den der *Maximierer* garantieren kann,
- β den *niedrigsten* Wert, den der *Minimierer* noch zulässt.

Sobald $\beta \leq \alpha$ gilt, kann der entsprechende Teilbaum „abgeschnitten“ werden, da er keine bessere Entscheidung mehr ermöglichen kann bzw. weil in einem anderen Zweig bereits ein mindestens ebenso gutes Ergebnis garantiert wird.

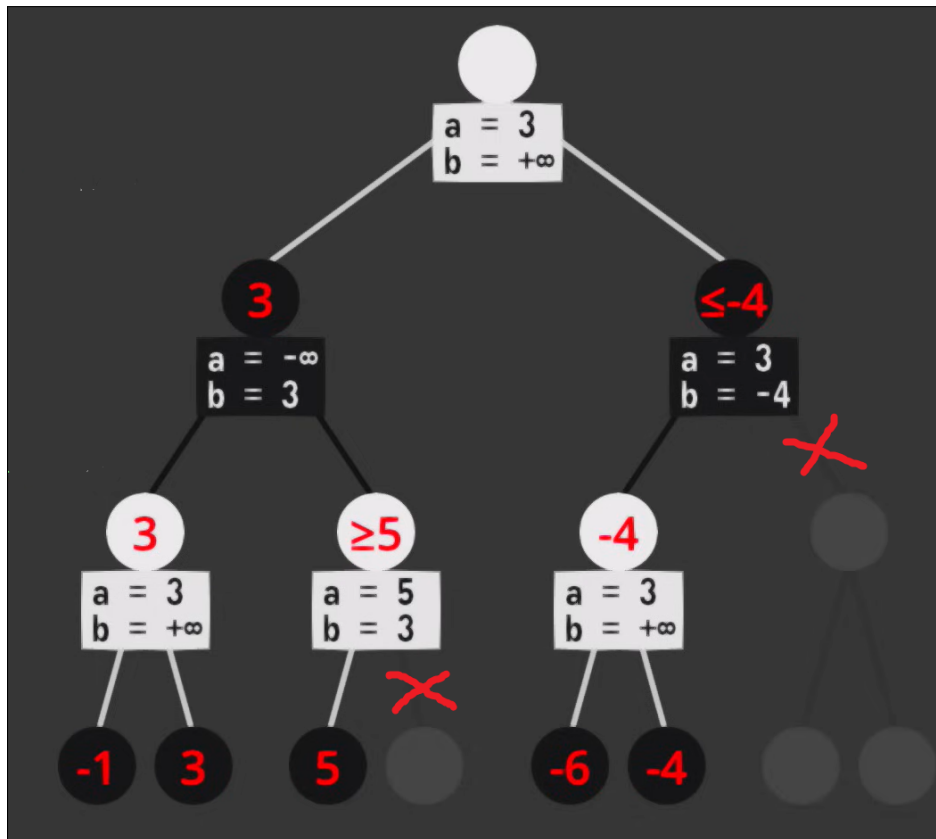


Abbildung 3.2: Alpha-Beta Suchbaum mit eliminierten Zweigen *Quelle:[6]*

3.2.1 Pruning-Beispiel

Die obige Abbildung verdeutlicht, wie eine Eliminierung eines Teilbaums (*Pruning*) abläuft. Zunächst wird der linke Teilbaum durchlaufen und die Blätter ausgewertet:

- Erstes Blatt: Wert -1 . Ist $-1 > \alpha$? **Ja!** $\Rightarrow \alpha = -1, \beta = +\infty$.
- Zweites Blatt: Wert 3 . Ist $3 > \alpha$? **Ja!** $\Rightarrow \alpha = 3, \beta = +\infty$. Nun wird geprüft, ob $\beta \leq \alpha$ gilt: $\infty \leq 3$? **Nein!** \rightarrow Kein Pruning im linken Teilbaum.

Anschließend überträgt der Algorithmus die Werte $\alpha = 3$ und $\beta = +\infty$ an den Elternknoten und untersucht den rechten Teilbaum:

- Erstes Blatt im rechten Teilbaum: Wert 5 . Da $5 > \alpha$ gilt, wird $\alpha = 5$ gesetzt. Jetzt gilt $\beta \leq \alpha$: $3 \leq 5$? **Ja!** \rightarrow *Pruning*: Der gesamte rechte Teilbaum wird abgeschnitten, da Schwarz im linken Teilbaum bereits eine bessere Option (Wert 3) garantiert hat.

Dieser abgeschnittene Zweig wird nicht weiter untersucht, weil der Minimax-Algorithmus annimmt, dass der Gegner stets den bestmöglichen Gegenzug wählt.

4 Evaluationsfunktion

Die Bewertungsfunktion ist ein hauptsächlich was die Spielstärke von Stockfish ausmacht. Sie ordnet jeder Schachstellung eine numerische Bewertung zu, die angibt, wie vorteilhaft die Stellung für Weiß oder Schwarz ist. Ein Wert von 0.0 signalisiert eine ausgeglichene Stellung. Positive Bewertungen sprechen für einen Vorteil von Weiß, negative Werte auf einen Vorteil für Schwarz. Diese Bewertung dient der Engine als Orientierung, um im Rahmen der Suche vielversprechende Stellungen zu erkennen und gezielt weiterzuverfolgen. Es handelt sich dabei um eine statische Bewertung – das bedeutet, dass die Stellung ohne Vorausberechnung zukünftiger Züge beurteilt wird; die eigentliche Suche übernimmt der Minimax-Algorithmus mit Alpha–Beta–Pruning.

Die Bewertung stützt sich nicht allein auf das reine Materialverhältnis auf dem Brett. Es fließen auch strategische und taktische Faktoren in die Bewertung ein. Dazu gehören unter anderem [3]:

- **Material:** Anzahl und Wert der verbleibenden Figuren
- **Mobilität:** wie viele sinnvolle Züge den Figuren zur Verfügung stehen
- **Königssicherheit:** wie gut der eigene König geschützt ist
- **Bauernstruktur:** Anordnung der Bauern und mögliche Schwächen wie isolierte oder doppelte Bauern
- **Raumkontrolle:** welche Seite mehr Einfluss auf zentrale oder strategisch wichtige Felder ausübt
- **Figurenkoordination:** wie effektiv die Figuren zusammenarbeiten und sich gegenseitig unterstützen

Außerdem nutzt Stockfish sogenannte **Piece–Square Tables (PSQTs)**[8] – Tabellen mit vordefinierten Werten für jede Figur auf jedem einzelnen Feld des Schachbretts. Für jede Figurart existiert eine eigene 64-Werte-Tabelle, die einen Bonus bzw. Strafe je nach Position auf dem Brett vergibt. So erhalten beispielsweise Springer einen Bonus, wenn sie zentral platziert sind, Türme profitieren von offenen Linien, und Könige werden im Mittelspiel in den Ecken als sicherer bewertet, während sie im Endspiel für eine aktive Rolle zentralisiert werden sollen. Die Engine kombiniert Bewertungskriterien für Mittelspiel und Endspiel gleitend, abhängig davon, wie viel Material noch auf dem Brett ist. Beispielsweise die Königssicherheit ist im Mittelspiel besonders wichtig, während im Endspiel eher die Aktivität des Königs zählt. Dabei wird nicht zu einem festen Zeitpunkt zwischen Mittel- und Endspiel umgeschaltet, sondern ein fließender Übergang geschaffen – je weniger Figuren noch auf dem Brett sind, desto stärker fließt die Endspielbewertung ein.

$$\text{Eval} = \frac{\text{phase} \times \text{ MiddlegameScore} + (1 - \text{phase}) \times \text{EndgameScore}}{\text{totalPhase}}$$

Die obige Formel[11] verdeutlicht, wie Stockfish eine dynamische Mischung aus Mittelspiel- und Endspielbewertung realisiert. Hierfür werden zwei Teilauswertungen – `MiddlegameScore` und `EndgameScore` – herangezogen, wodurch der Übergang vom Mittel- zum Endspiel abgebildet wird. Über viele Jahre hinweg basierte Stockfish vollständig auf handgeschriebene Bewertungsfunktionen, wie zuvor beschrieben. Mit der Einführung von **NNUE (Efficiently Updatable Neural Networks)** im Jahr 2020 hat sich dies grundlegend gewandelt[10]. Stockfish nutzt seither ein NNUE-Modell, das

- auf Millionen hochwertiger Schachstellungen trainiert wurde,
- Bitboard-Features effektiv in den Suchprozess integriert.

5 Schach KI Implementierung

5.1 Ansatz mit Convolutional Neural Networks

Im Rahmen eigener Recherchen zu Convolutional Neural Networks (CNNs) habe ich festgestellt, dass diese Architekturen gegenüber klassischen vollständig verbundenen Netzstrukturen (Fully Connected Networks) in vielen Hinsichten überlegen sind. CNNs sind aufgrund ihrer Fähigkeit, lokale räumliche Merkmale wie *Ecken* und *Kanten* zu erfassen, besonders gut für bildbasierte Aufgaben geeignet wie beispielsweise das Erkennen von handgeschriebenen Ziffern[7]. Diese Erkenntnis hat mein Interesse daran geweckt, inwiefern solche **räumlichen Merkmale** – etwa strukturelle Muster und lokale Abhängigkeiten – auch auf das **Schachbrett** und die dortige Stellungserkennung übertragbar sind. Insbesondere wollte ich untersuchen, ob CNNs in der Lage sind, schach-strategische Muster auf dem Brett (z. B. Angriffe, Deckungen, Drohungen) ähnlich effektiv zu erfassen wie Formen in einem Bild.

5.2 Problembeschreibung

Das vorgestellte CNN-Modell hat die Aufgabe, den nächsten optimalen Schachzug vorherzusagen – es handelt sich um ein überwachtes **Klassifikationsproblem mit mehreren Klassen**. Die Eingabe ist ein $9 \times 8 \times 8$ -Tensor, der den aktuellen Spielzustand abbildet – inklusive Figurenpositionen, legaler Zugoptionen und Zugfarbe. Jeder mögliche Zug wird auf einen eindeutigen Index im Aktionsraum abgebildet. Das Modell gibt einen dieser Indizes als Vorhersage zurück – also die wahrscheinlichste Zugklasse aus einer fest definierten Menge.

$$f : R^{9 \times 8 \times 8} \longrightarrow \{0, 1, \dots, K - 1\}, \quad \text{wobei } K \text{ die Anzahl der Klassen ist.}$$

5.3 Datensatz

Der Datensatz[2], der für das Training des Modells verwendet wurde, stammt von der Open-Source-Plattform **Kaggle**. Der ursprüngliche Umfang beläuft sich auf rund **6,25 Mio.** Schachpartien, deren teilnehmende Spieler ELO-Wertungen zwischen **700** und **3100** aufweisen. Es wurden ausschließlich Partien von Spielern mit einer ELO-Wertung über **2000** ausgewählt. Dadurch reduzierte sich die Datenmenge auf etwa **883 376** Partien.

5.4 Schachbrett Repräsentierung

Wenn der Datensatz, der aus einer Historie von Schachzügen innerhalb eines Spiels besteht, als Eingabe für das Netzwerk dienen soll, muss man sich eine geeignete Repräsentation überlegen, da ein neuronales Netzwerk bekanntlich mit Matrizen und nicht mit bloßen Zeichenketten arbeitet. Als Grundlage lässt sich – ähnlich wie bei den Bitboards von Stockfish – das Schachbrett in einer 8×8 -Matrix mit sechs Kanälen enkodieren, wobei jeder Kanal

einer bestimmten Figurenart (z. B. Springer, Läufer, Bauer, König, Dame, Turm) zugeordnet ist. Die Figuren von Weiß werden durch eine 1 markiert, die von Schwarz entsprechend durch eine -1 . Damit ist die Figurenkonstellation des Schachbretts zu einem bestimmten Zeitpunkt strukturiert abgebildet. Um der Eingabe zusätzliche Merkmale einer Schachstellung zu verleihen, wurden drei weitere Kanäle hinzugefügt:

- **Herkunftszug-Kanal:** Dieser Kanal markiert alle Ausgangsfelder legaler Züge. Im Fall des Zugs $e2xc3$ wäre beispielsweise das Feld $e2$ mit einer 1 belegt.
- **Zugziel-Kanal:** Dieser Kanal markiert die Zielfelder aller legalen Züge. Im oben genannten Beispiel wäre also das Feld $c3$ als eines der Zielfelder mit einer 1 markiert.
- **Zug-Indikations-Kanal:** Das gesamte 8×8 -Feld ist mit 1 belegt, wenn Weiß am Zug ist, sonst mit -1 , wenn Schwarz am Zug ist.

```

1 def createBoardRep(self, board):
2     """
3     Erzeugt eine mehrschichtige 8x8-Darstellung des Schachbretts.
4     Weisse Figuren: 1, schwarze: -1.
5     Zusätzliche Layer: legale Zugquellen, Zugziele, Spielerfarbe am Zug (1/-1).
6     Rueckgabe: np.array mit Form (9, 8, 8).
7     """
8     pieces = ['p', 'r', 'n', 'b', 'q', 'k']
9     layers = []
10
11     for piece in pieces:
12         layer = [[0 for _ in range(8)] for _ in range(8)]
13         for square, piece_obj in board.piece_map().items():
14             piece_type = piece_obj.symbol().lower()
15             if piece_type == piece:
16                 row = 7 - (square // 8)
17                 col = square % 8
18                 layer[row][col] = -1 if piece_obj.color == chess.BLACK else 1
19
20         layers.append(layer)
21
22     from_layer = np.zeros((8, 8), dtype=int)
23     for move in board.legal_moves:
24         from_square = move.from_square
25         row = 7 - (from_square // 8)
26         col = from_square % 8
27         from_layer[row][col] = 1
28     layers.append(from_layer)
29
30     # "To" squares layer
31     to_layer = np.zeros((8, 8), dtype=int)
32     for move in board.legal_moves:
33         to_square = move.to_square
34         row = 7 - (to_square // 8)
35         col = to_square % 8
36         to_layer[row][col] = 1
37     layers.append(to_layer)
38
39     # Turn indicator layer
40     turn_layer = np.full((8, 8), 1 if board.turn == chess.WHITE else -1,
41                           dtype=int)
42     layers.append(turn_layer)
43
44     return np.stack(layers) # Shape: (9, 8, 8)

```

Quellcode 5.1: Funktion zur Encodierung eines Schachbretts

Die Funktion `createBoardRep()` erzeugt die für das Modell erforderliche Repräsentation. Als Eingabe erhält sie ein `chess.Board`-Objekt (python-chess).

5.5 Zug-Index Konversion

Da das grundlegende Problem ein mehrklassiges Klassifikationsproblem ist, benötigen wir einen Weg, von einer Matrix-Schachstellung zu einer Integer-Indexzahl zu konvertieren und umgekehrt. Das Modell muss seinen Label-Move in einen Index konvertieren, sodass der Output einer Klasse entspricht. Der menschliche Nutzer des Modells braucht einen Weg,

die Ausgabe des Modells – den Integer-Index – einem Schachzug zuzuordnen. In meiner Implementation habe ich durch alle Züge jeder Partie im Datensatz iteriert und unique Einträge im Nachschlag-Dictionary für jeden Zug erstellt. Das bedeutet, dass ein möglicher Zug möglicherweise nicht im Dictionary enthalten ist und somit nie vom Modell als Vorhersage ausgegeben wird. Diese Tatsache hat außerdem zur Folge, dass:

- das Modell in Gefahr ist zu overfitten, indem es nicht generalisiert, sondern die meist-gespielten Züge wiedergibt: Der große Aktionsraum, die dieses Lookup-Dictionary darstellt, begünstigt diese Gefahr
- das Ungleichgewicht der Klassen wird kritisch – einige Züge (z. B. Bauernvorstöße oder Rochaden) können sehr häufig vorkommen, während seltene taktische Züge unterrepräsentiert sein können.

5.5.1 UCI-Format

Zu beachten ist außerdem, dass der gesamte Datensatz ins **UCI-Format (Universal Chess Interface)** konvertiert wurde. Während die **algebraische Notation** (z. B. „Sf3“) die relative Figur und ihren Zielort beschreibt, verwendet UCI **absolute Koordinaten** für Start- und Zielfeld. Der erwähnte Springerzug würde im UCI-Format beispielsweise als g1f3 dargestellt, falls der Springer auf g1 stünde. Diese Konvertierung bringt jedoch eine Herausforderung mit sich: Züge von Spezialfiguren (wie Springer, Läufer etc.) lassen sich im UCI-Format nicht leicht von Bauernzügen unterscheiden, da keine explizite Figurenangabe erfolgt. Dies birgt die Gefahr, dass das Modell während des Trainings verstärkt **Bauernzüge bevorzugt**, weil diese zahlenmäßig dominieren und keine Unterscheidung nach Figurentyp erfolgt.

5.6 Architektur

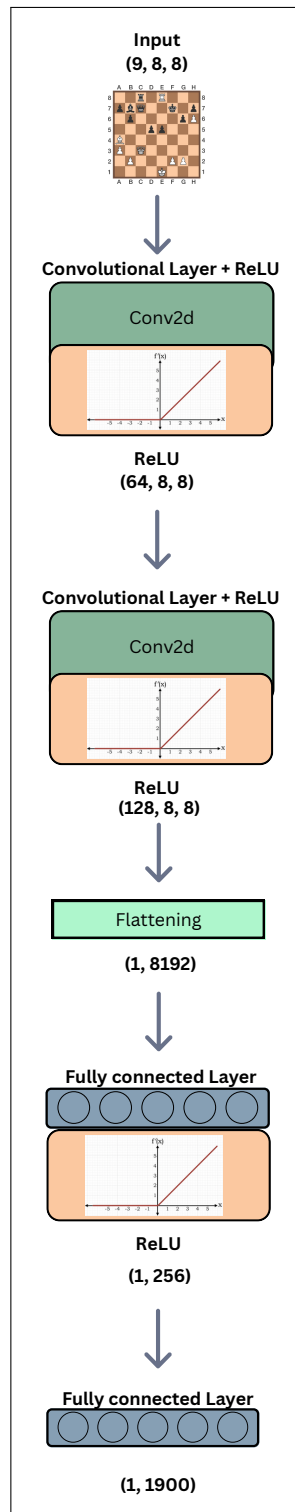


Abbildung 5.1: Forward-Pass des Basic Modells *Quelle:selbsterstellt*

5.7 Evaluation

6 Fazit

Im Großen und Ganzen hat mir dieses Studienprojekt viel technisches Wissen vermittelt. Ich durfte tief in die Welt der Schach-Engines eintauchen und war beeindruckt, auf welche kreative Weise in diesen Communities Lösungen entwickelt werden. Es fasziniert mich, dass sich etwas so Greifbares wie ein physisches Schachspiel digital abbilden und für komplexe Rechenzwecke nutzen lässt. Das Projekt erstreckte sich über insgesamt zwei Semester, da ich zu Beginn feststellen musste, dass mir die erforderlichen Kenntnisse im Bereich des Maschinellen Lernens fehlten. Diese Lücke konnte ich während einer Projektpause mithilfe der „Machine Learning Specialization“ von Andrew Ng auf Coursera schließen. Ich wünschte mir, dass entsprechende ML Lehrinhalte auch im Studiengang Softwaretechnik stärker verankert wären – besonders vor dem Hintergrund der rasanten Entwicklungen im Bereich der KI. Obwohl das von mir entwickelte Modell in keiner Weise mit Stockfish in Sachen Spielstärke konkurrieren kann, war die Implementierung und Evaluation eine außerordentlich lehrreiche Erfahrung.

An dieser Stelle möchte ich meinen herzlichen Dank an Prof. Dang richten – für seine wertvollen Anregungen und seine Geduld im Verlauf dieses Projekts.

Literaturverzeichnis

- [1] Ameye, Alexander. *Writing a chess engine in C++*. <https://ameye.dev/notes/chess-engine/>. [letzter Zugriff: 15-Mai-2025]. März 2022.
- [2] Arevel. *Chess Games*. <https://www.kaggle.com/datasets/arevel/chess-games>. [letzter Zugriff: 15-Mai-2025].
- [3] *Evaluation*. <https://www.chessprogramming.org/Evaluation>. [letzter Zugriff: 15-Mai-2025].
- [4] Hors, Analog. *Magical Bitboards and How to Find Them: Sliding move generation in chess*. <https://analog-hors.github.io/site/magic-bitboards/>. [letzter Zugriff: 15-Mai-2025]. Sep. 2022.
- [5] *How Many Possible Moves Are There In Chess?* <https://www.chessjournal.com/how-many-possible-moves-are-there-in-chess>. [letzter Zugriff: 14-Mai-2025].
- [6] Lague, Sebastian. *Algorithms Explained – minimax and alpha-beta pruning*. <https://www.youtube.com/watch?v=l-hh51ncgDI>. [letzter Zugriff: 15-Mai-2025].
- [7] LeCun, Yann et al. „Gradient-Based Learning Applied to Document Recognition“. In: *Proceedings of the IEEE* 86.11 (1998), Seiten 2278–2324. DOI: 10.1109/5.726791.
- [8] *Piece-Square Tables*. https://www.chessprogramming.org/Piece-Square_Tables. [letzter Zugriff: 15-Mai-2025].
- [9] Stockfish-Docs. <https://official-stockfish.github.io/docs/stockfish-wiki/Home.html>. [letzter Zugriff: 14-Mai-2025].
- [10] Stockfish-Docs. *NNUE*. <https://official-stockfish.github.io/docs/nnue-pytorch-wiki/docs/nnue.html>. [letzter Zugriff: 15-Mai-2025].
- [11] *Tapered Eval*. https://www.chessprogramming.org/Tapered_Eval. [letzter Zugriff: 15-Mai-2025].
- [12] Turochamp. <https://en.wikipedia.org/wiki/Turochamp>. [letzter Zugriff: 14-Mai-2025].
- [13] Wikipedia-Contributors. *Minimax*. <https://en.wikipedia.org/wiki/Minimax>. [letzter Zugriff: 15-Mai-2025].