Einleitung

Die Herausforderung, einen Computer zum Schachspielen zu befähigen, beschäftigte bereits Alan Turing und seine Studenten. 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.

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 diese Problematik bewältigen und welche faszinierenden Algorithmen und Heuristiken sie dabei verwenden. Es werden auch eine Architektur einer einfachen Schach-KI vorgestellt und umgesetzt, um einen näheren Einblick in die Funktionsweise und Programmierung einer Schach-KI zu geben.

Funktionsweise von Stockfish

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, eine Bewertungsfunktion, Bitboard-Repräsentation und heuristische Verfahren.

Schachbrett Repräsentation durch Bitboards

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-Brettes entspricht. Diese Darstellung ermöglicht extrem schnelle bitweise Operationen (AND, OR, XOR etc.), die die Grundlage der Evaluationsfunktion bilden.

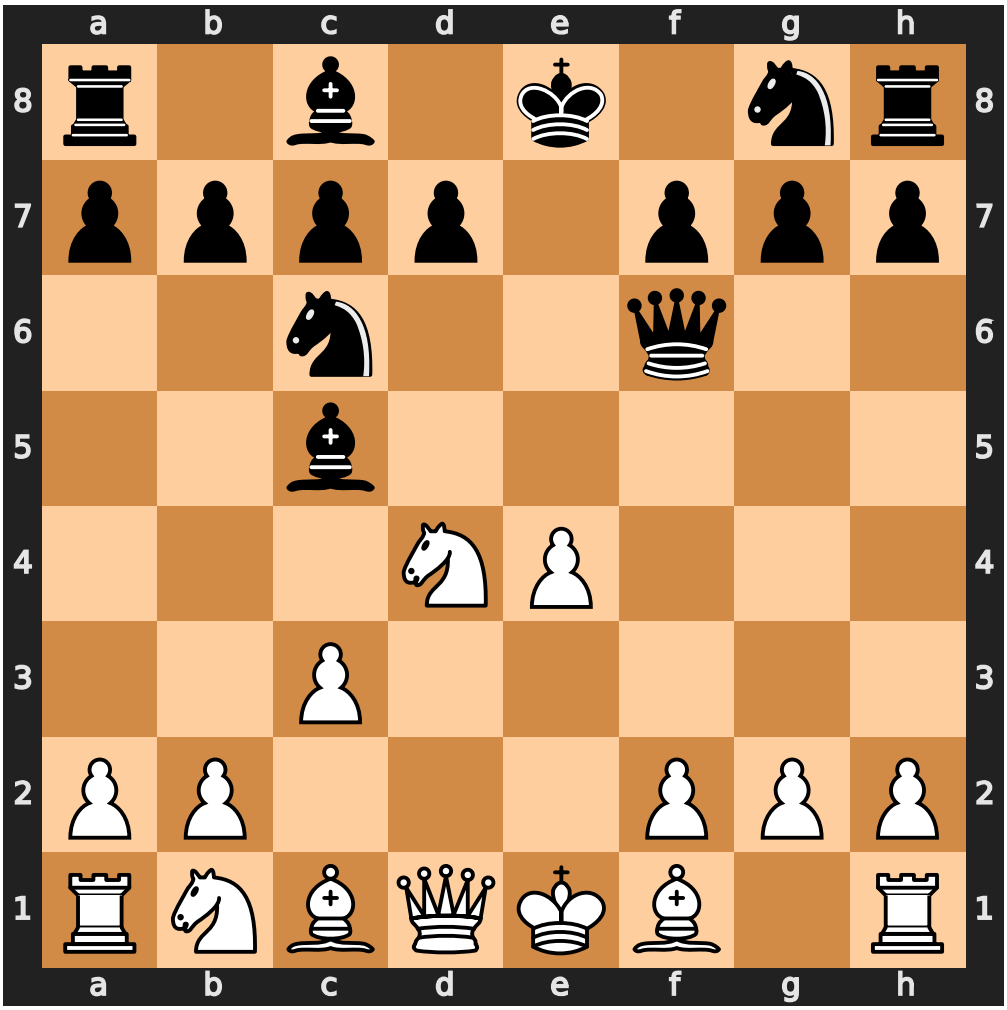

Abbildung 1.1: Schachbrett-Stellung

image source: [Analog Hors - Magical Bitboards and How to Find Them: Sliding move generation in chess]

Beispiel:
Die Position der Bauern in der obigen Abbildung ließe sich durch ein Bitboard folgendermaßen darstellen:

```
. . . . . . . .
1 1 1 1 . 1 1 1
. . . . . . . .
. . . . . . . .
. . . . 1 . . .
. . 1 . . . . .
1 1 . . . 1 1 1
. . . . . . . .
```

Spielfigur Unterscheidung

Für Zwecke der Schachzug-Generierung unterscheidet man zwischen sogenannten „springenden Figuren“ (engl. Leaping Pieces) und „gleitenden Figuren“ (engl. Sliding Pieces). Diese Unterscheidung basiert darauf, dass sich diese beiden Figurentypen in ihren Bewegungsmöglichkeiten und Einschränkungen deutlich unterscheiden.

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 können

```
//leeres Spielfeld
U64 attacks, knights = 0ULL;

// platziert Springer auf dem Brett
set_bit(knights, square);

// Springer Züge lassen sich mithilfe von Bit-Shifts generieren
attacks = (((knights >> 6)  | (knights << 10)) & ~FILE_GH) |
          (((knights >> 10) | (knights << 6))  & ~FILE_AB) |
          (((knights >> 15) | (knights << 17)) & ~FILE_H)  |
          (((knights >> 17) | (knights << 15)) & ~FILE_A);
```

Erklärung zu der Berechnung:

Der Springer bewegt sich in der Form eines 'L'. Das bedeutet beispielsweise, dass der Shift **>> 17** einer Verschiebung um 2 Zeilen (16 Bit) plus 1 Feld nach rechts (1 Bit) entspricht. Ähnlich funktioniert der Shift **<< 10** – er steht für eine Verschiebung um eine Zeile nach oben (8 Bit) und um zwei Felder nach rechts (2 Bit).

Code-Auszug aus: Writing a chess engine in C++

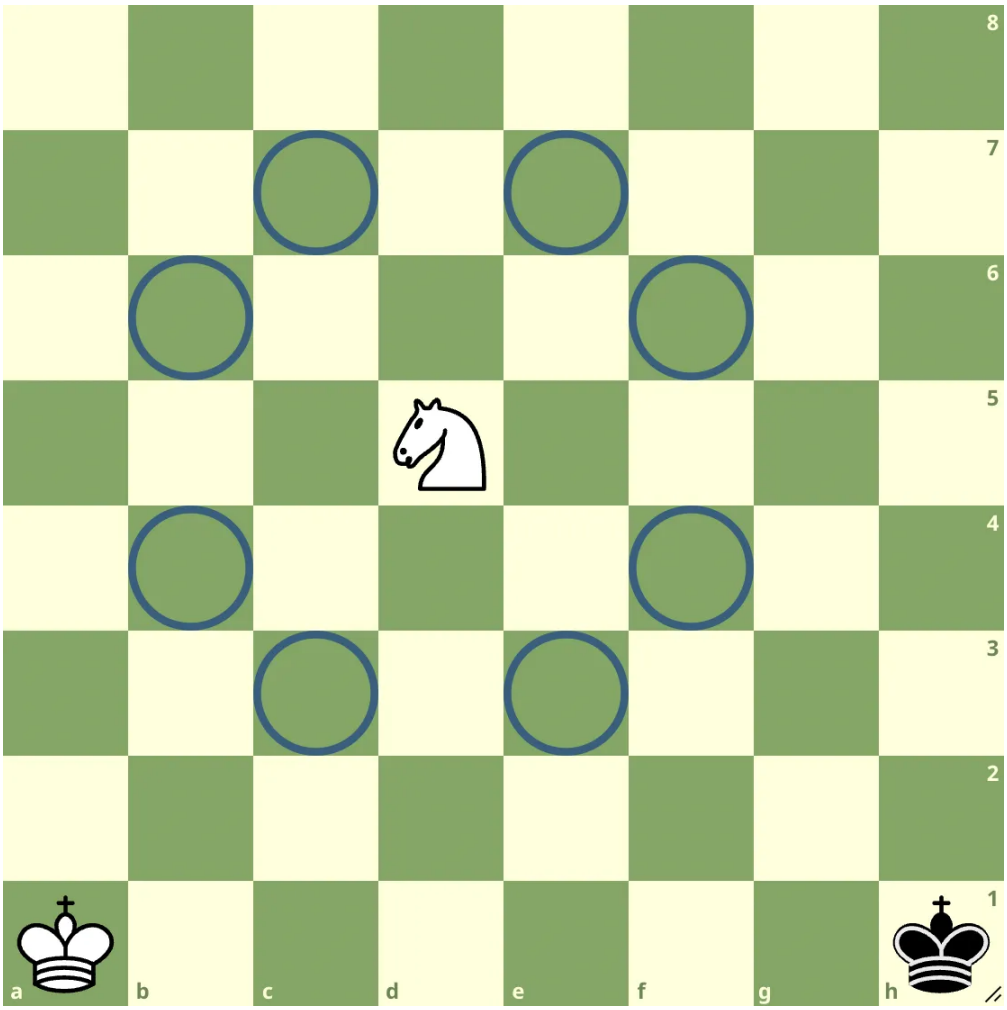

Abbildung 1.3: Das Bewegungs-Muster des Springers

Gleitende Figuren (Läufer, Turm, Dame)

Problematik der Gleitenden Figuren:
Gleitende Figuren wie Läufer, Türme und Damen können mehrere Felder in gerader Linie ziehen - aber nur so lange, bis sie auf einen "Blocker" treffen (entweder eine eigene oder eine gegnerische Figur).

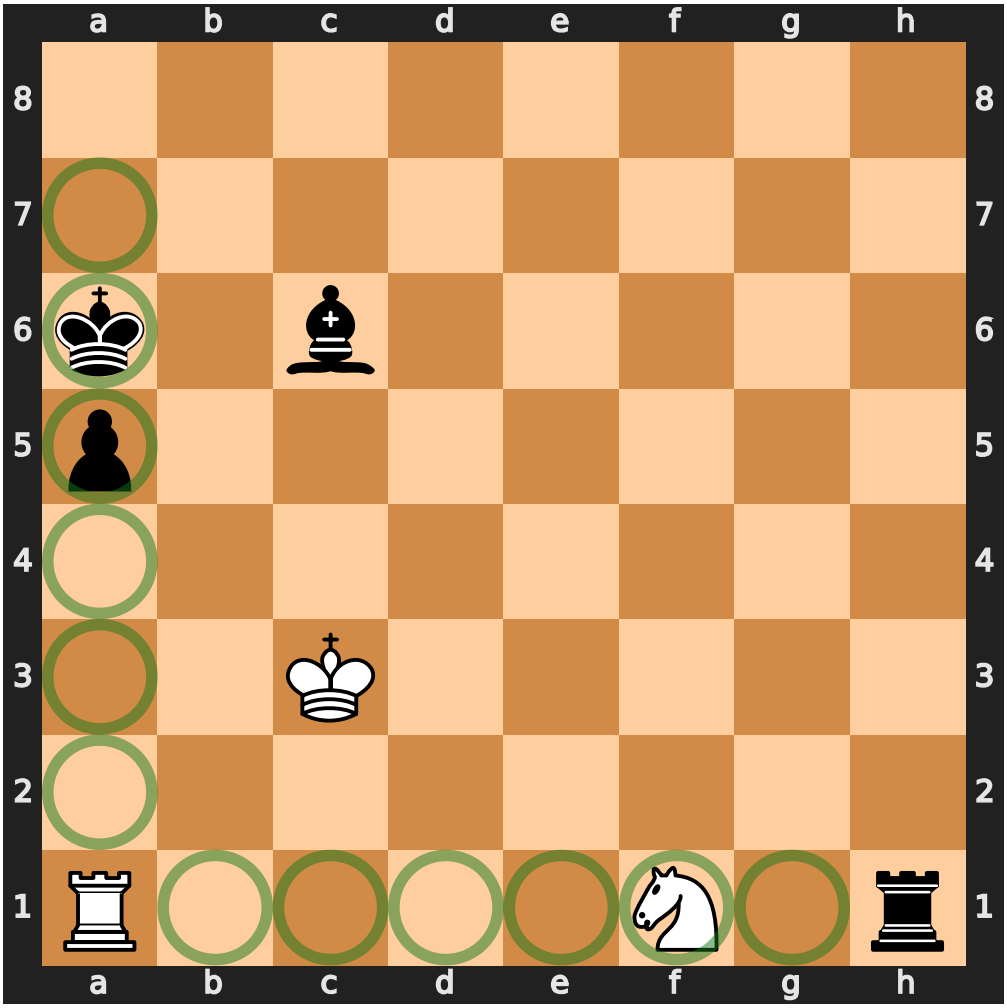

Abbildung 1.2: Der a1-Turm wird auf a5 und f1 in seiner geraden Bewegungslinie blockiert

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 kann.

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 Turm in Abbildung 1.4 lassen sich als folgende Bitmaske kodieren:

```
. . . . . . . .
1 . . . . . . .
1 . . . . . . .
1 . . . . . . .
1 . . . . . . .
1 . . . . . . .
1 . . . . . . .
. 1 1 1 1 1 1 .
```

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 Blocker-konfigurationen 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:

```
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. `0x0000000000000010`
2. `0x0000000100000000`
3. `0x0001000000000000`

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.

Hash-Verfahren:

```
uint64_t blockers = // 64-Bit kodiertes Schachbrett, wobei nur die tiefsten 12 Bits gesetzt sind
uint64_t magic    = // vorgerechnete "magische" Zahl

                   // die Multiplikation bewirkt ein "Durchmischen" der gesetzten Bits
int index         = (blockers * magic) >> (64 - 12);
//der abschließender Shift liefert die 12 höchsten Bits
// Index ist somit eine Zahl zwischen 0 und 4095
```

Das Verfahren beweritk, dass eine kollisionsfreie Lookup-Tabelle entsteht. Die „magische“ Zahl wird im Vorfeld per Trial-and-Error berechnet und anschließend als Konstante verwendet.

Wie Stockfish Schachzüge generiert

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 Schachzug-Generierung erfolgt bei springenden Figuren über direkte Bitboard-Operationen, während für die gleitenden Figuren (Läufer und Türme) das zuvor beschriebene Magic Bitboards zum Einsatz kommt. Dieses erlaubt eine besonders effiziente Bestimmung der Angriffsflächen in konstanter Zeit O(1).

Mini-Max Algorithmus:

Die Spieltheorie bennent Spiele, bei denen die Summen der Gewinne und Verluste aller Spieler gleich null ist, als Nullsummenspiele.
Mini-Max ist eins der gängigen Algorithmen der bei Nullsummenspielen wie Schach oder Tic-Tac-Toe eingesetzt wird. Stockfish setzt Mini-Max ein um mögliche Züge als Knoten in einem Suchbaum darzustellen, und weist den Knoten mithilfe einer Bewertungsfunktion eine Zahl zu, die die Qualität des Zuges repräsentiert. Das Besondere an diesem Algorithmus ist, dass er

nicht nur den aktuell besten eigenen Zug sucht, sondern auch berücksichtigt, welcher Zug den Gegner in seinen zukünftigen Optionen möglichst stark einschränkt. Somit übernimmt ein Spieler die Rolle des *Maximierers*, der darauf abzielt, Züge mit dem höchstmöglichen Bewertungswert zu erreichen. Der Gegenspieler wird im Gegensatz zum *Minimierer* und versucht, den niedrigstmöglichen Wert zu erzwingen – also die für den Maximierer ungünstigste Fortsetzung. Dadurch wird nicht nur der eigene Vorteil gesucht, sondern zugleich der Gegner eingeschränkt.
In der Schachbewertung gilt dabei: Positive Werte deuten auf einen Vorteil für Weiß, negative Werte auf einen Vorteil für Schwarz.

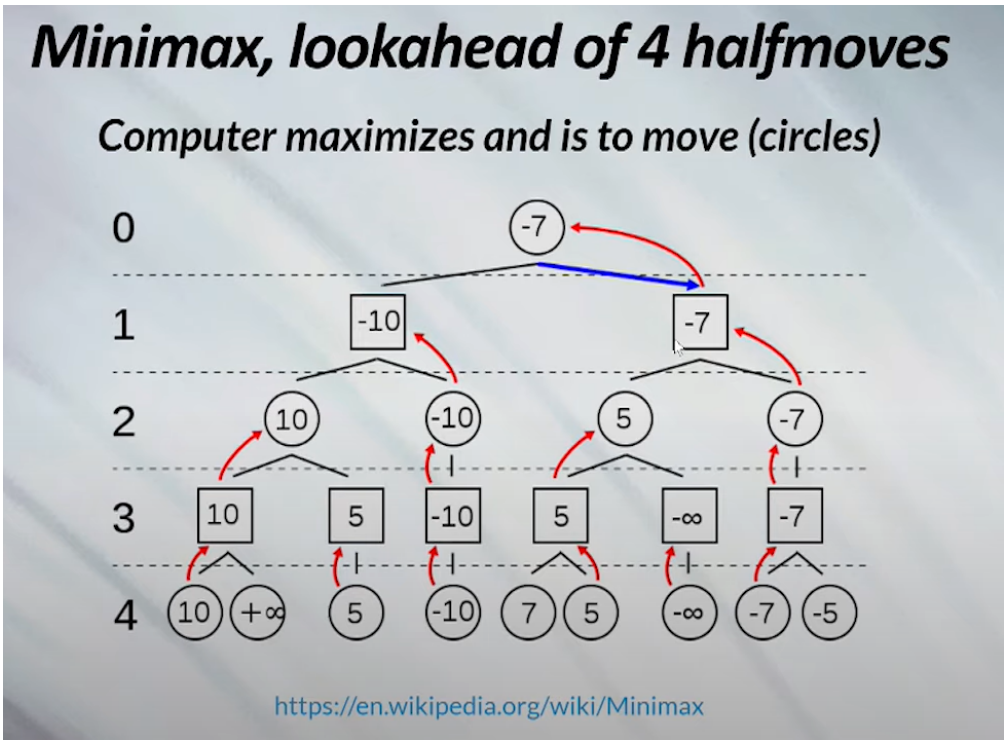

Abbildung 1.1: Minimax-Suchbaum mit Tiefe 4

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. **Abbildung 1.1** 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 **+∞** 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. Schwarz 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 30 möglichen Zügen pro Stellung aus und nimmt an, dass ein Programm 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 Studen
7	21,870,000,000	5 Tage

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.

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 **α (Alpha)** = `-∞` und **β (Beta)** = `+∞` initialisiert.
Im Verlauf der rekursiven Suche repräsentiert **Alpha** den aktuell **besten (höchsten)** Wert, den der **Maximierer** garantieren kann, während **Beta** den **niedrigsten** Wert darstellt, den der **Minimierer** noch zulässt.
Sobald Beta ≤ Alpha gilt, kann der entsprechende Teilbaum "abgeschnitten" werden, da er keine bessere Entscheidung mehr ermöglichen kann bzw. weil in einem anderen Zweig ihm ein besseres Ergebnis garantiert werden kann.

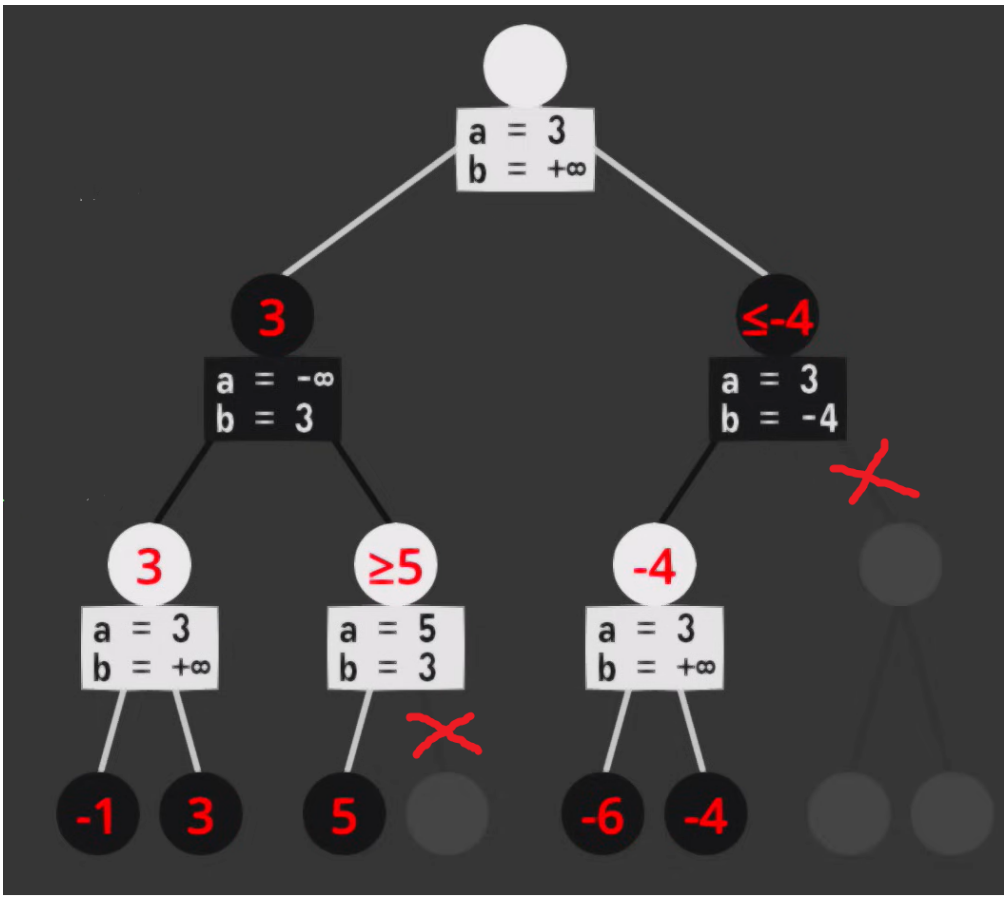

Abbildung 1.3: Alpha-Beta Suchbaum mit eliminierten Zweigen

- Das erste Blatt hat den Wert **-1** → ist `-1 > α`? **Ja!** → also wird `α = -1`, `β = +∞` bleibt unverändert.
- Dann wird das Blatt mit dem Wert **3** betrachtet → ist `3 > α`? **Ja!** → also wird `α = 3`, `β = +∞` bleibt ebenfalls unverändert.
  Jetzt wird geprüft, ob die Pruning-Bedingung `β < α` erfüllt ist → hier nicht erfüllt ❌, also kein Pruning.

Die Werte `α = -∞` und `β = 3` werden an den Vaterknoten übergeben, und der rechte Teilbaum wird untersucht.

- Das Blatt hat den Wert **5**, und da `5 > 3`, wird `α = 5` gesetzt.
  Jetzt ist `β < α` **erfüllt**: ✅
  Der Zweig kann abgeschnitten werden, da Schwarz im linken Zweig bereits eine bessere Option (**3**) garantiert hat.

Dieser Zweig wird also eliminiert, da der Minimax-Algorithmus davon ausgeht, dass der Gegner den bestmöglichen Gegenzug spielt.

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:

- **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) – 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.

```
Eval = (phase * MiddlegameScore + (1 - phase) * EndgameScore) / totalPhase
```

Über viele Jahre hinweg nutzte Stockfish keine neuronalen Netze, sondern setzte vollständig auf handgeschriebene Bewertungsfunktionen, wie sie zuvor beschrieben wurden.
Allerdings haben mittlerweile NNUE (Efficiently Updatable Neural Network) dies grundlegend verändert.
Das NNUE-Modell wird:

- auf Millionen hochwertiger Schachstellungen trainiert,
- mit Bitboards kombiniert und effizient in die Suche integriert.

Mein Ansatz:

Im Rahmen meiner eigenen Recherchen zu Convolutional Neural Networks (CNNs) habe ich herausgefunden, dass diese Architekturen der klassischen vollständig verbundenen Netzstruktur (fully-connected networks) effektiver sind – insbesondere bei der Erkennung handschriftlicher Ziffern.
Wie LeCun et al. (1998) gezeigt haben, sind .

CNNs sind aufgrund ihrer Fähigkeit, lokale räumliche Merkmale wie **Ecken und Kanten** zu erfassen, besonders gut für bildbasierte Aufgaben geeignet.

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, strategisch relevante Muster auf dem Brett (z. B. Angriffe, Deckungen, Drohungen) ähnlich effizient zu erfassen wie Formen in einem Bild.

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 9×8×8-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: \mathbb{R}^{9 \times 8 \times 8} \rightarrow {0, 1, \dots, K - 1}
${f: \mathbb{R}^{9 \times 8 \times 8} \rightarrow {0, 1, \dots, K - 1}}

Datensatz

Der Datensatz, den ich für mein Projekt verwende, stammt von der Open-Source-Plattform **Kaggle**. Der ursprüngliche Datensatz umfasst rund **6,25 Millionen Schachpartien**, wobei die **ELO-Wertungen** der beteiligten Spieler im Bereich von **700 bis 3100** liegen.
Da mein Ziel darin besteht, eine **stark spielende KI** zu entwickeln, habe ich den Datensatz gezielt gefiltert und nur Partien von Spielern mit einer ELO-Wertung über **2000** berücksichtigt. Durch diese Einschränkung wurde der Datensatz auf etwa **883.376 Partien** reduziert.

Schachbrett - Repräsentation

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 schonmal 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 durch 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.

```
    def createBoardRep(self, board):
    """
    Erzeugt eine mehrschichtige 8x8-Darstellung des Schachbretts.
    Weiße Figuren: 1, schwarze: -1.
    Zusätzliche Layer: legale Zugquellen, Zugziele, Zug am Zug (1/-1).
    Rückgabe: np.array mit Form (9, 8, 8).
    """


        pieces = ['p', 'r', 'n', 'b', 'q', 'k']
        layers = []
```

```
        for piece in pieces:
            layer = [[0 for _ in range(8)] for _ in range(8)]
            for square, piece_obj in board.piece_map().items():
                piece_type = piece_obj.symbol().lower()
                if piece_type == piece:
                    row = 7 - (square // 8)
                    col = square % 8
                    layer[row][col] = -1 if piece_obj.color == chess.BLACK else 1

            layers.append(layer)

        from_layer = np.zeros((8, 8), dtype=int)
        for move in board.legal_moves:
          from_square = move.from_square
          row = 7 - (from_square // 8)
          col = from_square % 8
          from_layer[row][col] = 1
        layers.append(from_layer)

        # "To" squares layer
        to_layer = np.zeros((8, 8), dtype=int)
        for move in board.legal_moves:
          to_square = move.to_square
          row = 7 - (to_square // 8)
          col = to_square % 8
          to_layer[row][col] = 1
        layers.append(to_layer)

        # Turn indicator layer
        turn_layer = np.full((8, 8), 1 if board.turn == chess.WHITE else -1,
dtype=int)
        layers.append(turn_layer)

        return np.stack(layers)  # Shape: (9, 8, 8)
```

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, den unser Nachschlag-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.

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.

Architektur

Für die Problemstellung werden zwei KI-Architekturen von CNN-Netzen vorgestellt:

```
def __init__(self, num_classes):
        super(ChessCNNModule, self).__init__()
        # conv1 -> relu -> conv2 -> relu -> flatten -> fc1 -> relu -> fc2
        # shape of input data is (9,8,8)
        self.conv1 = nn.Conv2d(9, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(8 * 8 * 128, 256)
        self.fc2 = nn.Linear(256, num_classes)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
```

```
        x = self.relu(self.conv2(x))
        x = self.flatten(x)
        x = self.relu(self.fc1(x))
        x = self.fc2(x)  # Output raw logits
        return x
```

```
class ChessCNNModule(nn.Module):
    def __init__(self, num_classes):
        super(ChessCNNModule, self).__init__()

        # Standard convolutions for local feature extraction
        self.conv1 = nn.Conv2d(9, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)

        # Dilated convolutions for larger receptive field
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=2, dilation=2)  #
dilation=2
        self.conv4 = nn.Conv2d(256, 256, kernel_size=3, padding=4, dilation=4)  #
dilation=4

        # Global Average Pooling instead of flatten
        self.gap = nn.AdaptiveAvgPool2d(1)  # Reduces (B, 256, 8, 8) -> (B, 256, 1,
1)

        # Fully connected layers with gradual dimension reduction
        self.fc1 = nn.Linear(256, 512)
        self.fc2 = nn.Linear(512, 768)
        self.fc3 = nn.Linear(768, num_classes)

        # Dropout for regularization
        self.dropout = nn.Dropout(0.3)

        # Activation
        self.relu = nn.ReLU()

        # Batch normalization for better training stability
        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)
        self.bn4 = nn.BatchNorm2d(256)

        # Initialize weights
        nn.init.kaiming_uniform_(self.conv1.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.conv2.weight, nonlinearity='relu')
        nn.init.kaiming_uniform_(self.conv3.weight, nonlinearity='relu')
```

```
        nn.init.kaiming_uniform_(self.conv4.weight, nonlinearity='relu')
        nn.init.xavier_uniform_(self.fc1.weight)
        nn.init.xavier_uniform_(self.fc2.weight)
        nn.init.xavier_uniform_(self.fc3.weight)

    def forward(self, x):
        # Standard convolutions
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))

        # Dilated convolutions for long-range dependencies
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.relu(self.bn4(self.conv4(x)))

        # Global Average Pooling
        x = self.gap(x)  # (B, 256, 1, 1)
        x = x.view(x.size(0), -1)  # (B, 256)

        # Fully connected layers with dropout
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)  # Output raw logits
        return x
```

Anforderungen:

- zitieren
- quellen angeben

für Dang wichtiger:
"dass es methodisch gut ist", muss nicht unbedingt "funktionieren"
→ Es braucht eine bewertung mittels train/test

Modelevaluierung:
CNN Architektur erklären können
→ weitere Conv2d ebenen einführen und Resultat vergleichen (vlt. 5-6 weitere Ebenen)

Evaluation:

- Train/test Split durchführen → Generralisierung beurteilen!!
- Overfitting / Underfitting berücksichtigen
- Loss funktionen Metriken notieren

Ausarbeitung:

soll ca. 20 Seiten am Ende befassen. Am Ende ergebnisse vorstellen!
Mit Folien, Vorgehensweise, Ergebnisse
ca. 15 Min Vortrag vorbereiten

- Stockfish teil vervollständigen

im Anschluss in der Ausarbeitung auf mein eigenes Engine eingehen
und wie **gut** es funktioniert → Metriken!