Das Meistern von chinesischem Schach mit autodidaktischem Reinforcement Learning

Simon Ma (15)[1]†

[1]Schillerschule Hannover, Ebellstr. 15, 30629 Hannover, Germany

Projektbetreuerin: Birgit Ziegenmeyer
Institution: Schillerschule Hannover
Thema des Projektes:
Die Entwicklung eines autodidaktischen Reinforcement Learning Algorithmus erweitert mit einem hochperformanten Suchalgorithmus um chinesisches Schach zu meistern
Fachgebiet: Mathematik/Informatik
Wettbewerbssparte: Jugend forscht
Bundesland: Niedersachsen
Wettbewerbsjahr: 2023

† Email Adresse: simon.mama07@gmail.com

Inhaltsverzeichnis

1. **Kurzfassung**

Xiangqi, chinesisches Schach, ist ein unvorstellbar komplexes Spiel, dessen Meisterung Jahrzehnte dauert - für einen Menschen. CheapChess ist meine Xiangqi-Applikation, mit Fokus auf einer selbstlernenden KI, die sich um die Frage dreht, wie sich die ressourcenintensiven existierenden Programme optimieren und sich komplexe Probleme kostengünstig lösen lassen. Zentrale Schwerpunkte lagen neben der Entwicklung eines funktionsfähigen Xiangqi Environments vor allem auf dem Entwurf und der Optimierung eines autodidaktischen Reinforcement Learning Algorithmus und dessen Erweiterung mit einem hochperformanten Suchalgorithmus, alles mit limitierter Hardware. Dafür entwarf ich neue Konzepte und es entstand ein flexibler, starker Spieler.

2. Einleitung

? Bereits für die Väter der Informatik wurde Computer Schach zur Herausforderung. Babbage, Turing und Shannon entwickelten schon in den frühen 1950ern Algorithmen und Hardware, um das menschliche Verständnis über die Domäne auszubauen. Schach wurde schnell zu einer der größten Herausforderung für Generationen von Forschern der künstlichen Intelligenz.

Die meisten Schach-Programme, z.B. Stockfish, basieren dabei auf einer Kombination von hocheffizienten Suchalgorithmen, domänenspezifischen Optimierungen und handgefertigten Evaluationsfunktionen, welche Jahre lang von Experten optimiert wurden.

Mit Deepminds AlphaZero führten 2017 innovative Konzepte und scheinbar unendliche Ressourcen zu einem der größten Durchbrüche in der KI-Forschung. Denn anstelle der herkömmlichen Evaluationsfunktionen lernte ein Convolutional Neural Network mithilfe eines raffinierten Reinforcement Learning allein durch Spiele gegen sich selbst Schach, Shogi und Go und schlug in jedem Spiel den jeweilige Weltmeister. Nur ist das Programm nicht öffentlich zugänglich und wohl kaum jemand besitzt die 5000+ Tensor Processing Units (TPUs), die Deepmind zur Verfügung standen.

Das grundsätzliche Ziel von CheapChess ist die Entwicklung eines flexiblen und anpassbaren Trainingspartners Anpassung benötigter Ressourcen auf normale Konsumenten, die Förderung von Zugänglichkeit zu innovativen Konzepten wie AlphaZero. Anders als Stockfish und AlphaZero fokussiere ich mich dabei jedoch spezifisch auf Xiangqi (chinesisches Schach). Xiangqi erfordert wegen seiner extremen Komplexität besondere Achtung auf Effizienz und durfte noch keine intensive Forschung genießen.

So entwarf ich eine Python Xiangqi Applikation und entwickelte einen effizienten selbstlernenden Reinforcement Learning Algorithmus ergänzt von hochoptimierten Such-Algorithmen. Ich erschuf eine Kombination aus den innovativen Konzepten von AlphaZero und den traditionellen Methoden von Stckfish, die beim Training beobachtet und zum Spielen herausgefordert werden kann.

Meine Arbeit soll einerseits eine radikale Reduktion der benötigten Hardware ermöglichen, andererseits den Stärken einer Methode erlauben, für die Schwächen der anderen zu kompensieren und darüber hinaus zu neuen Einblicken in die taktische Welt von Xiangqi führen.

3. Einführung zu Xiangqi

Xiangqi, auch bekannt als chinesisches Schach, ist ein strategisches Brettspiel, das tief in der chinesischen Kultur verwurzelt ist und seit dem neunten Jahrhundert überall in China gespielt wird. Seinen historischen Ursprung findet das Spiel schon um 200 v.Chr. , als sich zwei Dynastien vor einem der bedeutendsten Kämpfe der chinesischen Geschichte gegenüberstanden. Die Figuren sollen die zwei Armeen darstellen und der horizontale Barren in der Mitte des Feldes ist ein Symbol für den Fluss, der die Armeen auf dem Schlachtfeld trennte. Das Spiel ist in einigen Aspekten ähnlich wie Schach, weist aber auch Unterschiede auf. So wird es auf einem $9 \cdot 10$ Brett gespielt und es gibt andere Figuren. Das Brett besteht aus $9 \cdot 10$ Linien und Figuren stehen auf den Kreuzungen dieser Linien statt auf den Feldern. Um dem König herum ein Palast, welcher mit zusätzlichen Linien markiert ist 1.

Der Feldherr / König geht immer nur einen Schritt waagerecht oder senkrecht. Er darf den Palast nie verlassen und eine Rochade gibt es nicht. Die beiden gegnerischen Feldherren dürfen sich niemals ohne einen dazwischen stehenden Stein auf einer Linie gegenüberstehen. Der „Todesblick“ der Feldherrn verbietet dies, was besonders im End-

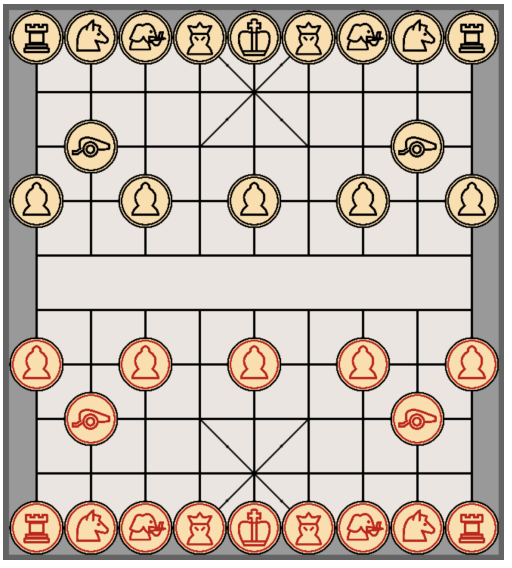

Abbildung 1. Die Standardaufstellung eines Xiangqi-Spiels, veranschaulicht mit dem GUI meiner Applikation

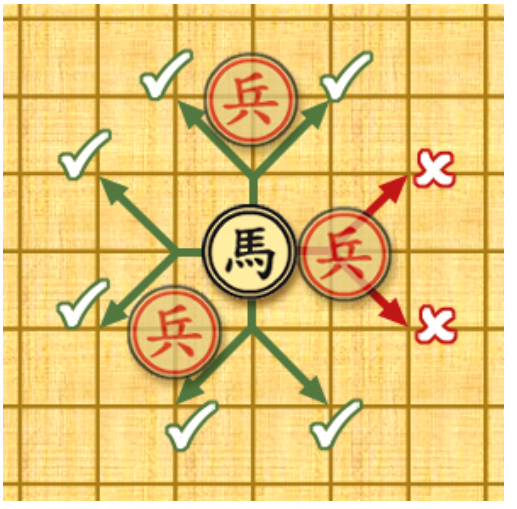

Abbildung 2. Züge des Pferdes

spiel genutzt werden kann, um ein Patt (in Xiangqi kein Remis, sondern ein Sieg) zu erzwingen.

Die Leibwächter / Berater gehen immer nur einen Schritt diagonal auf ein unmittelbar benachbartes Feld und dürfen den Palast ebenfalls nicht verlassen. Somit stehen beiden zusammen nur fünf Felder zur Verfügung, nämlich die Palastmitte und dessen vier Ecken.

Die Elefanten gehen genau zwei Schritte in diagonaler Richtung. Falls das zwischenliegende (übersprungene) Feld besetzt ist, wird der Zug geblockt. Darüber hinaus dürfen die Elefanten niemals den Fluss überqueren.

Die Pferde entsprechen im Wesentlichen den Springern des internationalen Schachs, können jedoch ebenfalls blockiert werden. Ein Pferd bewegt sich in seinem Zug in zwei Schritten: zuerst ein Feld waagerecht oder senkrecht in beliebiger Richtung und anschließend ein Feld diagonal. Das Pferd wird blockiert, wenn eine andere Figur auf dem zuerst zu betretenden Feld steht 2.

Der Turm / Wagen entspricht der den Türmen des europäischen Schachs.

Die Kanonen gibt es in internationalem Schach nicht. Wenn sie nicht schlagen, bewegen sie sich wie der Wagen. Zum Schlagen muss sich irgendwo zwischen dem gegnerischen Stein, und der Kanone genau ein anderer Stein befinden (Schanzenstein), der beim Schlagen übersprungen wird.

Die Soldaten / Bauer bewegen sich ein Feld nach vorne und können nicht zurückgehen. Anders als Bauer im internationalen Schach können sie aber nach Überquerung des Flusses ein Feld seitlich ziehen und schlagen nicht diagonal, sondern im herkömmlichen Zugmuster (senk- und waagerecht)

4. **Xiangqi-Engine**

4.1. *interne Brettdarstellung*

Die interne Brettdarstellung ist die möglichst effiziente Repräsentation des Spiels im Code. Die Figuren werden als Tupel repräsentiert, wobei Index 0 die Farbe und Index

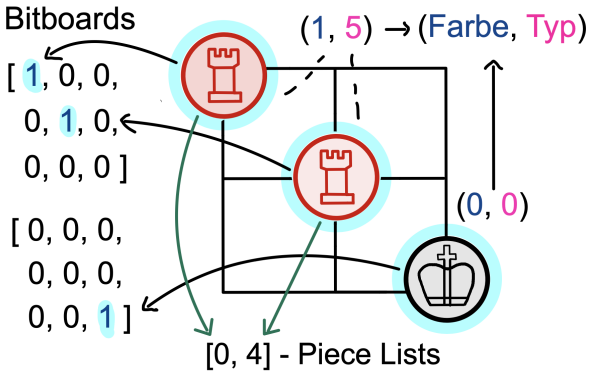

Abbildung 3. Simplifizierte Visualisierung der internen Brettrepräsentation

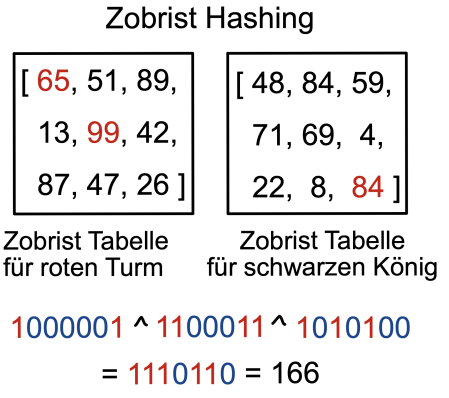

Abbildung 4. Simplifizierte Zobrist Hashing Visualisierung (Bezug auf 3)

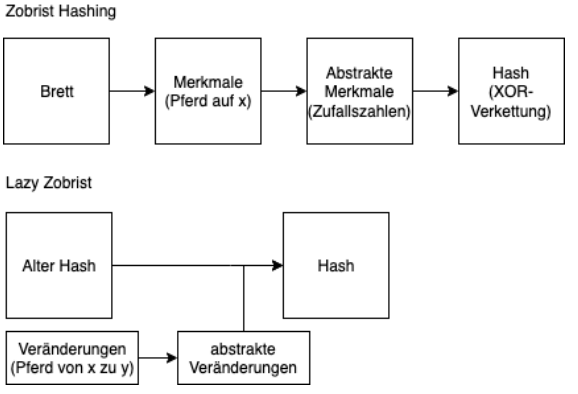

Abbildung 5. Unterschiedliche Vorgehensweisen von Zobrist Hashing vs. Lazy-Zobrist

1 den Typ bestimmt. Die darstellung als 5-bit Zahl funktionierte, erwies sich aber als deutlich langsamer. Das Brett $(9 \cdot 10)$ wird in drei verschiedenen Weisen dargestellt:

(i) Feldzentrisch als ein-dimensionale Liste mit Länge 90 für 90 Felder, 0 für leeres Feld, Tupel für Figur (intuitivste Darstellungsform).
(ii) Figurenzentrisch als drei-dimensionale Liste der besetzten Felder für jeden Figurentyp jeder Farbe 3.
(iii) drei-dimensionales array von Bitboards für jeden Figurentyp jeder Farbe. Form: $(2 \cdot 7 \cdot 90)$. Bitboards sind binäre Repräsentationen des Bretts, wobei eine 0 für ein leeres und eine 1 für ein besetzt Feld steht 3. Ich nutze sie als Eingabe für mein Neuronales Netzwerk.
(iv) Forsyth-Edwards-Notation (FEN) für das Laden eines Spielzustandes von einem string
(v) Zobrist-Hash-Key für die Kodierung des Spielzustandes in einen 64-Bit Integer

4.1.1. *Zobrist-Hashing und LaZo*

Für die folgenden KI-Suchalgorithmen ist es sehr wichtig, Spielzustände einfach und effizient darzustellen, z.B. für den schnellen Zugriff auf positionsspezifische Daten in einer Hash-Table. Ich fand heraus, dass der *Zobrist-Hashing* Algorithmus die Kodierung eines Spielzustands in einen einzigen n-Bit Integer ermöglicht. Ich wählte für die Größe des Hash-Schlüssels 64 Bits. Aufgrund der 2^{64} möglichen individuellen Hash-Werten

sind Hash-Kollisionen besonders unwahrscheinlich, was Zobrist-Hashing zu einer sehr zuverlässigen Methode macht.

Um Zobrist Hashing für Xiangqi zu implementieren, werden vorerst Zobrist-Tabellen initialisiert. Diese besteht aus Zufallszahlen für jede Farbe, jede Figur und jedes Feld. Dargestellt habe ich sie als dreidimensionales numpy-array bestehend aus zufälligen 64-Bit Integern. Danach werden die jeweilgen Werte in den Zobrist-Tabellen für jede Figur auf dem Brett mit die bitweisen XOR-Operation (Kontravalenz) verkettet. Die sich bewegende Seite wird ebenfalls berücksichtigt. Die Grundidee des Hashing-Algorithmus ist also die Abstrahierung der Merkmale des Bretts (mithilfe der Zufallszahlen der Zobrist-Tabellen) und die Nutzung dieser abstrakten Merkmale, um einen Hash-Schlüssel zu generieren.

Ich merkte hierbei, dass für die Generation des Hash-Schlüssels jedes mal über alle Figuren iteriert werden muss, obwohl sich für jeden Zug maximal die Position von zwei Figuren (*„X schlägt Y"*) verändern und alle restlichen Merkmale identisch bleiben. Durch diese Beobachtung entwarf ich ein Konzept, welches ich *Lazy-Zobrist (LaZo)* nenne, das anders als der traditionelle Zobrist-Hashing Algorithmus nicht auf Merkmale, sondern auf Veränderungen fokussiert ist. Dafür untersuchte ich Eigenschaften von Kontravalenzen und fand Folgendes heraus:

$$(A \veebar B) \veebar B = A \tag{4.1}$$

In anderen Worten ist es möglich, vorherige XOR-Operationen rückgängig zu machen. Relevant ist dies, da man somit bestimmte Merkmale vom Zobrist-Hash-Schlüssel entfernen kann. Auf die Optimierung vom Zobrist Hashing Algorithmus lässt es sich wiefolgt übertragen:

$$Z_a = Z_{a-1} \veebar T_{f_{a-1}} \veebar T_{f_a} \veebar T_{c_{a-1}} \tag{4.2}$$

Wobei Z für den Zobrist-Hash-Wert, T für die Zobrist Tabellen, a für einen Zeitpunkt des Spiels, f für die bewegte und c für eine ggf. geschlagene Figur stehen ($\veebar$ ist die mathematische Notierung der XOR-Operation). Bei jedem Zug wird die bewegte Figur f auf seiner ursprünglichen Position (Zeitpunkt $a-1$) vom Hash-Wert als Brett-Eigenschaft gelöscht und auf seiner neuen Position (Zeitpunkt a) hinzugefügt. Falls eine Figur c geschlagen wurde, wird diese ebenfalls vom Hash-Schlüssel entfernt.

LaZo vermeidet also die wiederholte Generation eines neuen Hash-Schlüssels für jede Position, ermöglicht stattdessen die Modifizierung eines existierenden Hash-Schlüssels 5 und sorgt für ein identisches Resultat mit deutlich weniger Berechnungen.

4.2. *Schwerpunkt: Zuggeneration*

Die Entwicklung eines Zuggeneration-Algorithmus ist die effiziente Übersetzung der Spielregeln in Code. Weil er von der KI innerhalb Sekunden Milliarden Male genutzt wird, ist exzellente Performance für die Leistung der KI essenziell. Da keine gute Python-Implementationen existieren, schrieb ich meinen eigenen Zuggenerationsalgorithmus, bestehend aus zwei Hauptkomponenten:

(i) Kodierung von Bewegungsmustern ohne Analyse der Legalität
(ii) Herausfiltern unerlaubter Züge

4.2.1. *Generation pseudo-legaler Zugmuster*

In der ersten Komponente werden sogenannte Move-Maps generiert, worin die grundsätzlichen Zugregeln kodiert sind. Diese Züge können in bestimmten Situationen aber gegen die Regeln verstoßen, weshalb sie nur *pesudo-legal* sind. Die Generation ist einmalig, weshalb die Peformance hier nicht ausschlaggebend ist. Wichtig ist nur, die

Zugregeln so darzustellen, dass sie möglichst effizient von der nächsten Komponente genutzt werden können.

Die Generation der Zugregeln ist gut durchdacht und umfasst viele Schritte, weshalb eine ausführliche Erklärung zu lang wäre. Die Grundideen sind in folgenden Abschnitten beschrieben.

Ich überlegte mir: Da das Brett eine ein-dimensionale Repräsentation besitzt, kann ein Skalarwert (ein-dimesional) als Offset genutzt werden, um zu einem anderen Feld zu gelangen. Für einen Schritt nach rechts wäre dieser z.B. 1, nach oben -9, diagonal nach rechts oben -8. Züge werden als Array repräsentiert, wobei die erste Stelle den Ursprung und die zweite Stelle das Ziel angibt. Von Feld 10 wäre ein Zug nach oben also [10, 10-9] oder [10, 1]. Aus diesen Offsets berechne ich dann die jeweiligen Pferd-Sprung Offsets und füge all diese einem Offset-Array hinzu. Diese stehen immer in der gleichen Reihenfolge und können mit einem Richtungs-Index in Bezug genommen werden.

Die Move-Map der Kanone und des Turms besteht hierbei aus einer Liste an Hash-Tabellen, wo der Listen-Index für das Feld steht, der Hash-Schlüssel den Richtungs-Index angibt und der dazugehörige Wert eine Liste aus möglichen Zielfeldern ist. Dafür wird für jedes Feld und für jede Richtung (N, O, S, W; Richtungs-Index 0 - 3) die Distanz zum Ende des Bretts berechnet, dann die Anzahl von Schritten in die jeweilige Richtung gegangen und alle Felder auf dem Weg als Ziel für den jeweiligen Richtungs-Index hinzugefügt.

Für das Pferd werden die vorher berechneten Pferd-Sprung-Offsets genutzt. Die Manhattan Distanz zwischen Start und Ziel muss exakt drei betragen um Sprünge zur anderen Seite des Bretts zu vermeiden. Da das Brett 1D ist, würde der Zug [9, 16] trotz eines validen Offsets (7) zur anderen Seite des Bretts Springen, was nicht erlaubt ist.

Die Elefanten Move Map wird mit einem Depth-First-Search (DFS) Algorithmus generiert, da kein einfaches Muster zwischen den erreichbaren Feldern besteht. Hierfür nutze ich ein Stack s und füge ein Start-Feld hinzu. Nun wird iterativ von s ein besuchtes Feld f entfernt und für dieses alle möglichen Ziele z an s angehängt. Währenddessen wird die Move Map an Stelle f mit z ergänzt. Dabei kann kein Ziel z_i seinen Ursprung f_i zu s hinzufügen. Das gleiche Prinzip wie bei dem Pferd gilt auch für den Elefanten, um unerlaubte Sprünge zu vermeiden.

Da die Leibwächter nur fünf Felder erreichen können, wird jeweils das Zentrum c des Palastes ausgewählt und ein Schritt diagonal in die Ecken e des Palastes gegangen. Für jede Ecke wird der Move-Map an Hash-Schlüssel c das Ziel e_i und an Hash-Schlüssel e_i das Ziel c hinzugefügt.

Für den König / General berechne ich die möglichen vertikalen Offsets für die drei Reihen und die horizontalen Offsets für die drei Spalten im Palast. Diese werden dann zu einem zweidimensionalen Offset-Vektor für alle neun Felder im Palast kombiniert und daraus die möglichen Züge berechnet.

Der Handlungsraum (en. Action Space) umfasst alle möglichen Züge.

4.2.2. *Legalitätsfilter*

Dieser Teil des Algorithmus hat die Aufgabe, unerlaubte Züge zu erkennen und zu entfernen. Er umfasst im Quellcode mehr als 800 Zeilen, besteht aus vielen Schritten und würde für eine genaue Erklärung zu viel Platz erfordern, weshalb ich diesen Teil ebenfalls abstrahiert erklären muss.

Mein erster Gedanke war es, für jeden Zug alle möglichen Antworten des Gegners zu generieren. Wenn eine Antwort davon den König schlägt, ist der Zug unerlaubt. Das Problem ist hierbei aber nicht nur die offensichtliche Ineffizienz, sondern vor allem die Ungenauigkeit - *Was ist, wenn diese Antwort selber unerlaubt ist?*

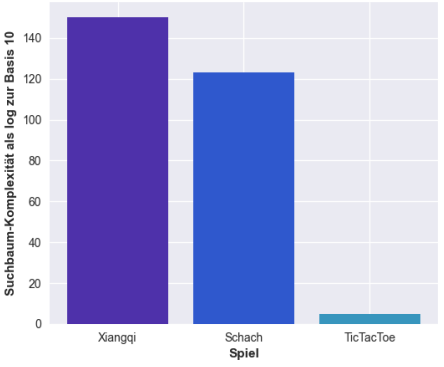

Abbildung 6. Suchbaum-Komplexität Vergleich

Spielstein	Bezeichnung	Wert
兵 卒	Soldat	1 Einheit
兵 卒	Soldat nach Überschreiten des Flusses	2 Einheiten
仕 士	Leibwächter	2 Einheiten
相 象	Minister/Elefant	2 Einheiten
傌 馬	Pferd	4 Einheiten
炮 砲	Kanone	4,5 Einheiten
俥 車	Wagen	9 Einheiten

Abbildung 7. Wert der Figuren

Da diese Komponente essenziell für die Performance aller folgenden Algorithmen ist und keinen Spielraum für Ineffizienzen und Ungenauigkeiten bietet, ist dies keine Lösung.

So plante, entwickelte und optimierte ich einen neuen Algorithmus, der Angriffsdaten generiert und mithilfe dieser die (un-)erlaubten Züge zu identifizieren. Es müssen dafür also Pins und Schachs erkannt werden und das Verhalten dieser Figuren unterschiedlich angepasst werden. Offensive Figuren, mögliche Bedrohungen müssen frühzeitig identifiziert werden um unnötige Angriffsdaten-Generation zu vermeiden. Diese Grundidee dieser Komponente besteht also aus drei Teilen:

(i) mögliche Bedrohungen erkennen
(ii) Angriffsdaten generieren
(iii) mit den Angriffsdaten unerlaubte Züge herausfiltern

Besondere Schwierigkeiten bereitete mir dabei die Kanone, da sie für äußerst skurrile Grenzfälle sorgen kann, mit denen ich vorher nicht gerechnet hatte und die sich nur durch unerwartetes Verhalten des Programms zeigen und nach viel Debugging und Troubleshooting lösen lassen. Erst durch die vielen Schwierigkeiten während der Entwicklung realisierte ich die wirkliche Komplexität des Spiels, welches ich anfangs stark unterschätzt hatte.

Trotzdem konnte ich die Performance vom Zugfilter un seine Präzision mit jeder Optimierung. Diese bestanden grundsätzlich immer aus der Integration von neuen Ideen und der geschickten Veränderung von Datenstrukturen oder vom Algorithmus.

5. **KI-Suche**

Die Suche in einer KI beschreibt den Prozess, unterschiedliche Züge zu spielen, diese zu bewerten und aus ihnen den besten Zug zu wählen. Hierfür hilft es, sich das Spiel als einen Baum-Datenstruktur mit Spielzuständen (en. state) $s_{root}, s_2, s_3...$ als Knoten vorzustellen. Diese Knoten werden von Zügen $a_1, a_2, a_3...$ als Zweige verbunden. Hierbei ist s_{root} der Startzustand und alle untergeordneten Knoten beschreiben Positionen, die aus Folge von den Zügen entstehen. Diese Baum Datenstruktur wird auch *Search Tree* bzw. *Such-Baum* genannt. Wenn das Endergebnis des Spiels für jeden Zug kennen würden, bestünden die Bewertungen nur aus Verlust, Unentschieden und Sieg und die KI müsste nur einen Zug in die Zukunft schauen, um den Besten zu finden. In der Praxis

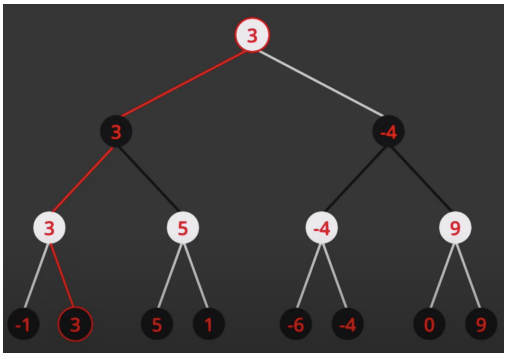

Abbildung 8. Einfache Visualisierung von Minimax. Weiß: max, schwarz: min

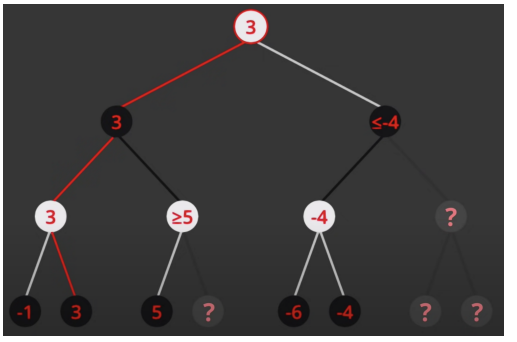

Abbildung 9. Der gleiche Suchbaum, mit Alpha-Beta-Pruning

ist kann jedoch aufgrund der unvorstellbar großen Such-Baum-Komplexität von Xiangqi (10^{150}) der exakte Wert einer Position nicht ermittelt werden, weshalb dieser geschätzt werden muss.

5.1. *SHEF*

Genau dafür nutze ich eine *Standard-Heuristic-Evaluation-Function (SHEF)*, die einen statischen Wert für jede Figur festlegt 7 und damit einen Spielzustand bewertet. *Piece-Square-Tables* implementierte ich, um diese Werte zu spezifizieren. Diese gehen von der Beobachtung aus, dass Figuren abhängig von ihrer Position unterschiedlich stark sind und geben demnach jeder Figur einen Feld-spezifischen Wert. Beispielsweise ist ein Turm vor dem gegnerischen Palast stärker als in einer Ecke. Eine großes Problem der SHEF, welches ich dabei erkannte, ist die absolute Abhängigkeit der Bewertung von den Positionen der Figuren und die fehlende Berücksichtigung der Brettstruktur, was zu großen Ungenauigkeiten führen kann (deshalb auch *heuristisch*).

5.2. *Quiescene-Suche*

Betrachtet man folgendes Beispiel: *„**X steht auf einem starken Feld**, kann aber im nächsten Zug geschlagen werden"*, wird schnell klar, wie dies zu Problemen führen kann. Weil die SHEF nur die Position des Pferdes berücksichtigt und nicht sieht, dass das Pferd bedroht wird, implementierte ich eine *Quiescene-Suche*, die eine Position so lange ausspielt, bis eine Position erreicht ist, wo keine Figuren geschlagen werden können. Somit sollen große Entscheidungsfehler behoben werden, indem nur Positionen ausgewertet werden, in denen der erste Teil des Beispiels zutrifft. Ein weiteres Problem bleibt aber. Im Fall *„**X steht auf einem starken Feld**, ist aber völlig umzingelt und damit stark eingeschränkt und weniger Wert"* , wird auch nach Einführung der Quiescene-Suche nur der erste Teil berücksichtigt und die Evaluation bleibt in vielen Fällen ungenau.

5.3. *MiniMax*

In meiner Recherche über einfache Spiel-KIs stieß ich auf den populären und weitverbreiteten MiniMax-Algorithmus. Dieser ist zwar simpel und ineffizient, bildet aber das konzeptionelle Fundament selbst für Programme wie Stockfish. Es wird angenommen, dass jeder Spieler an jeder Situation den optimalen Zug spielt. Das bedeutet, dass ein Spieler den erwarteten Gewinn laut der SHEF maximiert und der Gegner versucht, diesen zu minimieren. So werden die Blätter (unterste Knoten) des Such-Baums mit den Evaluationen der jeweiligen Positionen gekennzeichnet und die restlichen Knoten werden

rekursiv durch Auswahl des Zuges mit dem höchsten (für **max**) bzw. dem niedrigsten (für **min**) Gewinn gekennzeichnet. Ich habe eine elegantere Methode genutzt, wo jeder Spieler den besten erreichten Wert seines Gegners mit -1 multipliziert, denn ein guter Wert aus Perspektive des Gegners ist ein schlechter Wert für sich selbst. Somit kann auf unterschiedliche Operationen basierend auf dem Spieler verzichtet und immer die *max()* Funktion genutzt werden.

Ich implementierte diesen Algorithmus, erzielte jedoch nicht mein Ziel, hochperformante Algorithmen zu schreiben. Die Suche dauerte schon für drei Züge in die Zukunft rund zwanzig Sekunden. Dies lag daran, dass Minimax-Algorithmen eine exponentielle Zeit-Komplexität von $O(b^d)$ haben, wobei b der Ausdehnungsfaktor und d die Tiefe (en. depth) der Suche ist.

5.4. *Alpha-Beta-Pruning*

Ich fand heraus, dass Alpha-Beta-Pruning die Leistung des Minimax-Algorithmus deutlich verbessert, indem es unbrauchbare Zweige des Such-Baumes frühzeitig abschneidet. Dies wird erreicht, indem für jeden Knoten im Baum zwei Werte, α und β, verwendet werden. α repräsentiert den besten Wert, den der aktuelle Spieler erreichen kann, und β repräsentiert den besten Wert, den der gegnerische Spieler erreichen kann. Wenn $\alpha \geqslant \beta$ ist, wird der Rest des Zweigs abgeschnitten, da der Gegner den Knoten mit Wert a nicht wählen wird und der aktuelle Spieler keine Möglichkeit hat, ein besseres Ergebnis zu erzielen.

Die Verwendung von α und β ermöglicht es dem Algorithmus, schneller Entscheidungen zu treffen, indem unbrauchbare Zweige des Entscheidungsbaums und der damit verbundenen untergeordneten Baum-Struktur des Entscheidungsbaumes frühzeitig abgeschnitten werden. Dies führt zu einer signifikanten Reduzierung der Anzahl der Knoten, die untersucht werden müssen und damit zu einer deutlichen Leistungssteigerung. Die Untergrenze der Zeit-Komplexität liegt nur noch bei $\Omega(b^{d/2})$

5.5. *Zugsortierung*

Um die Anzahl der abgeschnittenen Zweige bei der Alpha-Beta-Suche zu maximieren, habe ich einen Zug Sortier-Algorithmus entwickelt. Dafür wird an jedem Knoten die Priorität jedes Zuges a anhand des Wertes der sich bewegenden Figur f und ggf. einer geschlagenen Figur c bestimmt:

$$v_a = c_a * m - f_a$$

Hierbei ist m eine Konstante, welche die Gewichtung des geschlagenen Figurenwertes kontrolliert. Ich fand heraus, dass die meisten Zweige abgeschnitten werden konnten, wenn $m \geqslant \frac{b}{w}$ ist (b steht für den höchsten Wert unter allen Figuren, w für den Schlechtesten). Somit hat selbst das Schlagen schlechter Figuren mit starken Figuren Vorrang über nicht-schlagende Züge. So wird z.B. ein Turm-schlägt-Bauern Zug vor einem Bauer-schlägt-Turm Zug erforscht, welcher wiederum Vorrang vor einem normalen Bauer-Zug hat.

5.6. *Multiprocessing und Parallelismus*

Um die Performance zu verbessern, verteilte ich den Algorithmus mit dem Multiprocessing Modul auf mehrere Prozessor-Kerne und konnte somit in acht Sekunden bis zu fünf Züge in die Zunkunft (*133 Millionen* Positionen) suchen.

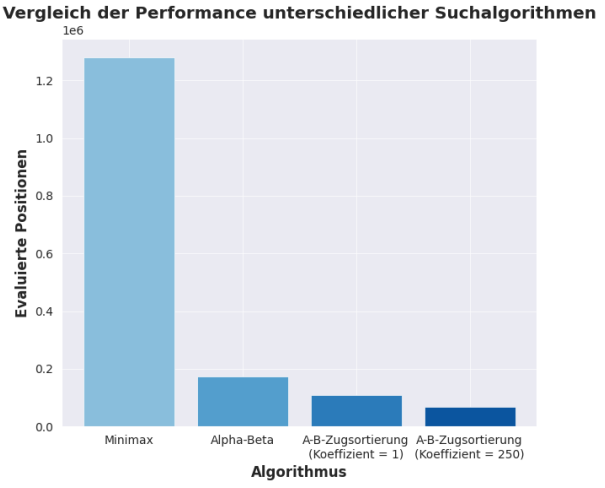

Abbildung 10. Evaluierte Positionen von jedem Algorithmus bei Suchtiefe von 4. Die Daten sind von jeweils 105 Suchen für jeden Algorithmus an unterschiedlichen Startpositionen gesammelt

6. Autodidaktisches Reinforcement Learning

Ich hätte die Alpha-Beta Suche noch lange weiter optimieren können, mir fehlte aber der Antrieb, weiterhin nur bereits existierende Konzepte zu verstehen und implementieren. Außerdem entdeckte ich während der Entwicklung entscheidende Schwächen der Alpha-Beta Suche.

Ich machte es mir demnach zum Ziel, einen weiteren großen Schritt zu gehen und dem Computer das Lernen zu Lehren.

Zu Beginn hatte ich vor, große Datenbanken an Spielaufzeichnungen von Profis zu nutzen und damit ein neuronales Netzwerk zu trainieren. Das Problem war aber, dass diese nicht existieren, weshalb der Algorithmus selber in der Lage sein muss, diese Trainingsdaten zu generieren. Somit soll auch verhindert werden, dass das neuronale Netzwerk nur lernt, Menschen zu imitieren und kein neues Verständnis über das Spiel erlangt und eine Art Kreativität entwickelt. Zur Verfügung stehende Mittel sind dabei nur die Spielregeln. In anderen Worten: Etwa wie wir Menschen soll ein Deep Convolutional Neural Network durch eine Vielzahl von Spielen gegen sich selbst, tabula rasa, Xiangqi meistern.

Ich wusste bereits AlphaZero von Deepmind, las den Forschungsbericht, schaute mir Präsentationen an und fand heraus, wie es AlphaZero gelang, mit einem generischen Reinforcement Learning Algorithmus nur durch Spiele gegen sich selbst mehrere strategische Brettspiele zu meistern. Dies beseitigte vor allem das Bedürfnis zu professionellem Wissen und menschlicher Arbeit, um weitere domänenspezifischen Optimierungen zu treffen und löste weitere Probleme einer Minimax-basierten Suche. Hierfür hat Deepmind 2017 einen Forschungsbericht über AlphaZero veröffentlicht, in dem die grundsätzliche Idee beschrieben wurde. Es ist aber zu beachten, dass die Architektur und genauere Informationen zum Trainings-Algorithmus von AlphaZero geheim sind und darüber hinaus anzunehmen ist, dass normale Hardware sie bei weitem nicht handhaben kann. Dieser Teil des Projekts erforderte mehrere Monate an Recherche und Kopfzerbrechen.

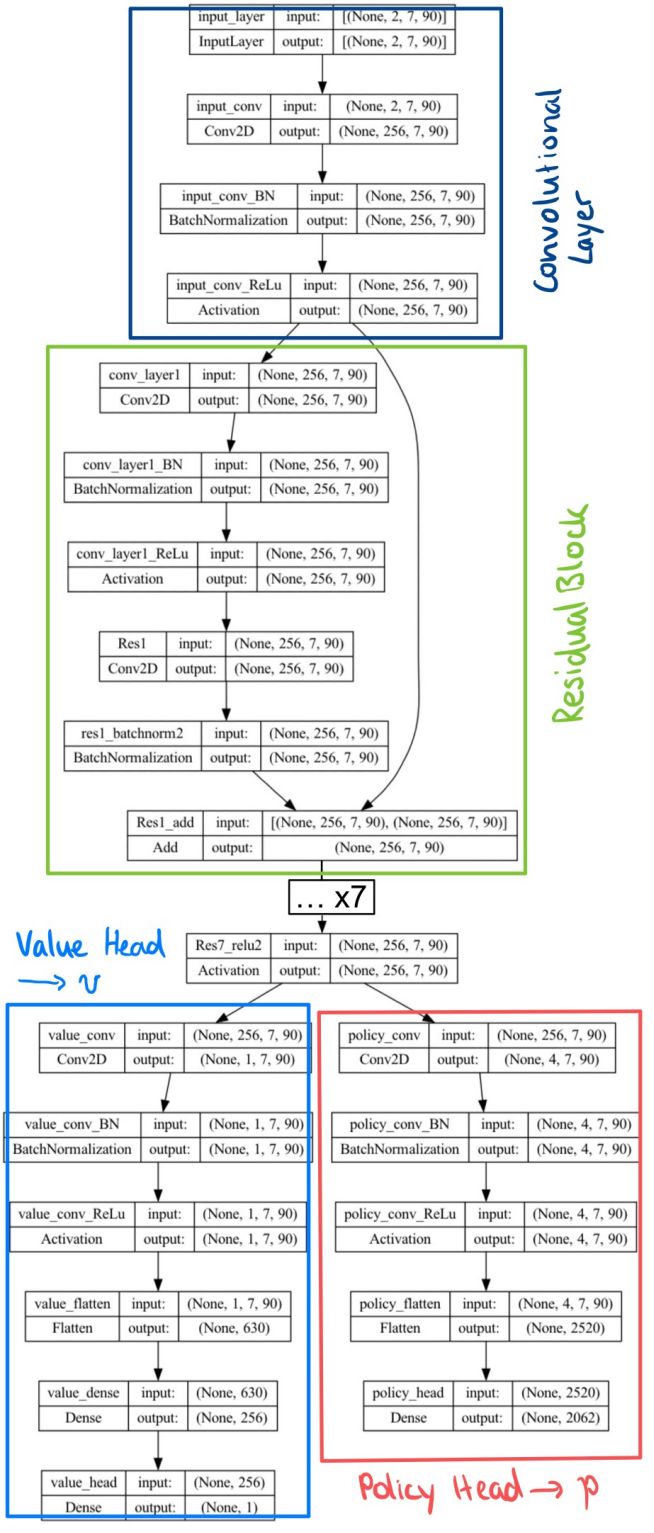

Abbildung 11. Neuronales Netzwerk visualisiert

6.1. *Modellarchitektur*

Um die bereits angesprochenen Probleme der SHEF zu lösen, entwarf ich ein Deep Convolutional Residual Neural Network mit tensorflow und keras, welches wie wir Menschen die Brettstruktur berücksichtigen soll und für normale Hardware gut geeignet ist. Hier war es wichtig, die Architektur möglichst klein zu halten, gleichzeitig aber so groß zu lassen, dass es lernfähig ist (*„So klein wie möglich, so groß wie nötig*). Mathematisch sieht es wiefolgt aus:

$$(p, v) = f_\theta(s)$$

Eingabe ist die Brettposition s, Ausgabe ein Vektor einer Wahrscheinlichkeitsverteilung p über alle Züge in der Action Space und ein Skalar v zwischen -1 (Niederlage) und 1 (Sieg), der das erwartete Ergebnis von Position s schätzt. Das Netzwerk lernt die optimalen Parameter θ. Es lernt in anderen Worten also, wie gut bestimmte Züge in einer bestimmten Position sind und den Wert der Position selbst zu schätzen. Meine Architektur besteht aus zwei Hauptkomponenten:

(i) Ein Convolutional Layer. Diese sind generell besonders gut für Erkennung räumlicher Hierarchien und Muster, ähnlich wie das menschliche visuelle System. Der Convolutional Layer nimmt den aktuellen Brettzustand als Eingabe und gibt eine Reihe von Merkmalen aus, die das Brett repräsentieren.
(ii) Ein Multi-Layer Perceptron (MLP), das die Ausgabe vom Concvolutional Layer als Eingabe nimmt und neben dem geschätzten Wert des Spielzustandes v eine Wahrscheinlichkeitsverteilung p über alle möglichen Züge ausgibt. Das MLP besteht aus sieben Residual Layers, einem *value head* (für den Wert v) und einem *policy head* (für die Wahrscheinlichkeitsverteilung p).

Residual-Schichten ermöglichen es dem Netzwerk statt der Ausgabe den Residual, also den Unterschied zwischen Eingabe und gewünschter Ausgabe, zu lernen. Dadurch wird das Problem des verschwindenden Gradienten behoben. Residual-Schichten erleichtern auch das Training sehr tiefer Netzwerke, da sie es ermöglichen, neue Schichten hinzuzufügen, ohne die Leistung zu verschlechtern.

6.2. *MCTS*

Der Monte-Carlo Tree Search (MCTS) Algorithmus wird aus gleichen Gründen wie beim Zuggenerator abstrahiert. Wichtig ist, dass die Suche eine neue, informierte und somit genauere Wahrscheinlichkeitsverteilung π mithilfe des aktuellen CNN f_θ generiert, welche dann genutzt werden, um die Parameter θ anzupassen. Ich habe mir erschlossen, dass MCTS auf der Idee basiert, den menschlichen Gedankenverlauf zu simulieren.

Der Algorithmus besteht aus vier Schritten: Auswahl, Erweiterung, Simulation und Backpropagation.

(i) Auswahl: Im ersten Schritt wird der Algorithmus durch den aktuellen Zustand des Spiels navigieren und den Zug auswählen, der die höchste geschätzte Siegchance hat.
(ii) Erweiterung: Sobald ein Blattknoten erreicht ist, wird dieser mit untergeordneten Knoten erweitert, die erlaubte zukünftige Zustände des Spiels repräsentieren.
(iii) Simulation: Der Algorithmus bestimmt dann die Wahrscheinlichkeit, dass der ausgewählte Zug zum Sieg führt. Hierfür wird die Wert-Ausgabe v vom Neuronalen Netzwerk genutzt.
(iv) Backpropagation: Schließlich wird das Ergebnis der Simulation entlang des im Suchbaum gegangen Weges zurückprogagiert, um die geschätzte Siegchance für jeden besuchten Knoten im Entscheidungsbaum zu aktualisieren.

6.3. *Augmentation von Trainingsdaten*

Nachdem ich das Grundprinzip des selbstlernenden Algorithmus implementiert hatte, bekam ich die Idee, die Bitboards und Züge immer an die Perspektive eines Spielers anzupassen. Dadurch wurde die Anzahl der Labels, also der Größe der CNN-Ausgabe verringert, da einige Züge nach perspektivischer Anpassung äquivalent sind, z.B. ist [89, 71] aus Perspektive eines Spielers äquivalent mit [0, 18] aus Perspektive des Anderen. Diese Methode ist menschennäher und sorgt für reduzierte Modellkomplexität, schnelleres Lernen und erhöhte Performance. Viel wichtiger ist aber, dass so allgemeinere Muster erkannt, standardisierte Trainingsdaten gesammelt und Verwirrung beim Training durch unterschiedliche Wertungen perspektivisch äquivalenter Züge vermieden werden können. Auch ist somit kein "side plane" mehr als Eingabe erforderlich, um die sich bewegende Seite anzugeben und die Eingabe ist komplett für die Erkennung von Mustern auf dem Brett optimiert. Ein weiterer wichtiger Vorteil ist die Augmentation von Trainingsdaten um 200%, weil für jede Position aus beiden Perspektiven ein Trainingsbeispiel gesammelt werden kann.

Ich erkannte außerdem, dass die Effektivität der Sammlung von Trainingsdaten um weitere 200% gesteigert werden kann, indem die Symmetrie des Bretts ausgenutzt wird, um die Bitboards entlang der y-Achse zu spiegeln. Somit kann jeder Spielzustand in ein Set aus vier Trainingsbeispielen umgewandelt werden.

6.4. *Training*

Traingsdaten werden durch Spiele gegen sich selbst mit MCTS gesammelt. Am Ende jedes Spiels wird jedes Trainingsset mit dem Endergebnis z ergänzt, sodass neben der verbesserten Wahrscheinlichkeitsverteilung π auch die Bewertung der Positionen genauer sind. Das Ziel vom Training ist es, die Differenz zwischen vorhergesagtem Ergebnis v und Endergebnis z zu minimieren und die Ähnlichkeit zwischen vorhergesagter Wahrscheinlichkeitsverteilung p und π zu maximieren, sodass das Netzwerk f_θ immer genauere Vorhersagen treffen kann. Hierfür nutze ich den SGD (Stochastic Gradient Descent) Optimierer und eine Verlust-Funktion l, welche mean-squared-error (MSE) für die Ergebnisse und categorical-cross-entropy für die Wahrscheinlichkeitsverteilung nutzt. Mathematisch notiert sieht diese wiefolgt aus:

$$l = (z - v)^2 - \pi^T \log p + c\|\theta\|^2$$

Das T steht für *Transpose*, eine Matrizen-Operation, die benutzt wird, um die Dimensionen für eine Matrizen-Multiplikation anzupassen.

7. **Alpha-Beta-Zero**

Nun habe ich bereits zwei komplett unterschiedliche Algorithmen implementiert.

Während sich AlphaZero also auf die in den letzten Jahren erst entdeckte Macht des maschinellen Lernens verlässt, bringt Stockfisch alte Techniken an ihre Grenzen. In meiner Recherche fand ich heraus, dass die unterschiedlichen Methoden für sehr viel Uneinigkeit und Zerspaltung zwischen KI-Forschern und Schach-Enthusiasten führte.

Ich empfand die ganze Aufregung um dieses Thema als äußerst unnötig. Vielleicht war es auch die fehlende Voreingenommenheit zusammen mit der Erkennung der Schwächen beider Methoden, welche mich zu der Entdeckung ihrer **Kompatibilität** führten, bevor ich sie überhaupt vergleichen konnte (einer meiner vorherigen Ziele).

Alpha-Beta-Zero ist der Teil von Algorithmus, in der ich Alpha-Beta-Suche und den von AlphaZero inspirierten selbstlernenden Reinforcement Learning Algorithmus kombiniere. Ich entwickelte mehrere Ideen, um dies zu tun und entschied mich am Ende für

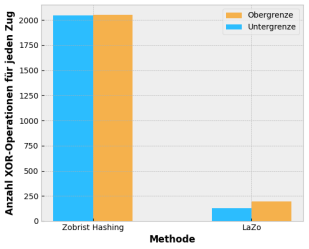

Abbildung 12. Vergleich der Berechnungen von Zobrist-Hashing und LaZo

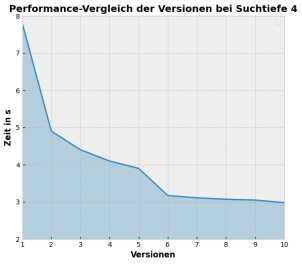

Abbildung 13. Performance-Vergleich unterschiedlicher Versionen des Zuggenerators bei Suchtiefe 4

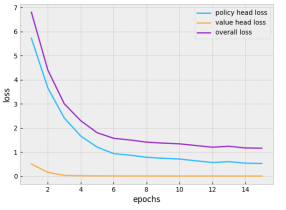

Abbildung 14. Die Loss-Werte im Laufe von 15 Trainings-Epochen mit 800+ Trainingsbeispielen

die effizienteste, einfachste und eleganteste Methode. Dabei führe ich beide Algorithmen parallel aus, ermittele einfach den ausgewählten Zug beider Algorithmen und addiere ihren Wert laut den Evaluationen beider Such-Algorithmen, um am Ende den Zug auszuwählen, der V maximiert.

$$V = \pi_a \cdot m + \beta_a \tag{7.3}$$

Die Evaluationen beider Algorithmen sind π für MCTS, β für die Alpha-Beta-Suche und a ist ein Zug. Der Koeffizient m kontrolliert dabei die Gewichtung der MCTS-Evaluationen und steht somit proportional zur Autonomie des Neuronalen Netzwerks, sodass der Algorithmus sich in einem frühen Stadium des CNN bei einem niedrigem m-Wert größtenteils auf die Alpha-Beta Suche verlässt. Im Gegenteil wird bei einem leistungsstarken CNN mit viel Training mithilfe eines hohen m-Werts nur bei offensichtlichen Entscheidungsfehlern vom MCTS, z.B. bei starkem vorhergesehenen Materialverlust oder einem Matt, eine Änderung des Zuges vorgenommen.

Alpha-Beta-Zero ist ein Hybrid aus zwei sehr unterschiedlichen Algorithmen, der die Schwächen einer Methode mit den Stärken der anderen ausgleicht.

8. **Ergebnisse**

Nach gut acht Monaten und unzählbar vielen Iterationen von Recherchieren, Nachdenken, Ausprobieren, Scheitern und wieder Recherchieren gelang es mir, erfolgreich eine Xiangqi-KI zu entwickeln, die einen selbstlernenden Algorithmus mit der traditionellen Alpha-Beta-Suche harmonisch ergänzt.

Dafür entwickelte ich selber eine Xiangqi-Applikation. Dabei entwarf ich eine neue Methode, um Zobrist-Hashing um bis zu 1600% zu verbessern 12. Mein Zuggenerator ist mittlerweile die schnellste Python-Implementation weltweit und kann bei Performance-Tests auf meinem Laptop bis zu 1.080.000 Blätter pro Sekunde im Suchbaum erreichen 13. Diese Performance-Tests (perft) werden durchgeführt, um die Leistung des Zuggenerators zu ermitteln und seine Genauigkeit durch den Vergleich mit dem Konsens anderer Programme zu überprüfen. Diese stimmen zu 100% überein.

Ich entwickelte einen domänenspezifisch optimierten Alpha-Beta-Suchalgorithmus, welcher in ca. acht Sekundenbis zu fünf Züge (133.000.000 Blattknoten) in die Zukunft schauen kann.

Im Reinforcement Learning Algorithmus konnten durch Anpassungen der Modellarchitektur-Komplexität die benötigte Hardware von 5000+ TPUs auf einen Laptop reduziert

werden. Die Modifizierung Eingaben des neuronalen Netzwerks erleichterte dies und erlaubte kombiniert mit der Ausnutzung von Symmetrien des Bretts eine Augmentation der Trainingsdaten um 400%. Die Generation von vier Trainingssets dauert dadurch nicht mehr als drei Sekunden und das Training verläuft reibungslos auf einer einzelnen GPU 14.

9. **Ausblick**

Das Projekt befindet sich derzeit immer noch im Entwicklungsstadium. Zurzeit arbeite ich an einem sogenannten Elo-System zur Evaluation meiner Alpha-Beta-Zero Methode. So kann die Spielstärke der KI gemessen und seine Entwicklung analysiert werden, damit ich anschließend die Hyperparameter für den Trainingsprozess optimieren kann.

Der Zuggenerator ist nach zahlreichen Optimierungen zwar schon hochperformant, besitzt aber noch weiteres Potenzial für Steigerungen der Performance durch Just-in-time-Kompilierung mithilfe von Bibliotheken wie numba (dann wäre es aber natürlich nicht mehr pures Python). Auch die Alpha-Beta-Suche kann weiter verbessert werden, etwa mit Methoden wie iterative deepening, transposition tables und der Optimierung vom Parallelismus.

Ein weiterer interessanter Schritt, den man zukünftig gehen könnte, ist die Modifizierung eines echten Bretts, sodass die KI mithilfe eines Microcontrollers ein eingebautes mechanisches Konstrukt unter dem Brett steuern kann, um die Entscheidungen der KI in echte Züge auf einem physischen Brett zu übertragen.

10. **Zusammenfassung**

Zusammengefasst konnte ich die benötigte Hardware innovativer Algorithmen auf normale Konsumenten anpassen, indem ich diesen mit einem hochoptimierten traditionellen Suchalgorithmus ergänzte, entwickelte dafür eine hochperformante Xiangqi-Applikation und entwarf neue Konzepte und Algorithmen. Dies war nur möglich, weil ich Dafür musste ich zahlreiche mathematische Konzepte verstehen und mein algorithmisches Denken an die Grenzen bringen.

Ich habe realisiert, dass neue Konzepte oft durch die Kombination von Neu und Alt, durch die Erweiterung von innovativen Konzepten und traditionell anerkannten Methoden, entstehen.

Ich lernte auch, dass diese neuen Konzepte nur durch kritisches Hinterfragen und intensiver Beschäftigung mit den herkömmlichen Methoden entsteht und oft sehr viel Mut und eine Bereitschaft zum Scheitern erfordert. Meine Perspektive zu Bugs und ungeplantem Verhalten des Programms änderte sich und ich fing an sie nicht als Scheitern, sondern als Versuche zu sehen.

Es war ein langer Prozess vom wiederholten Kopfzerbrechen und Scheitern, der mir beibrachte, dass das Verstehen von komplexen Themen manchmal vielleicht gar nicht so unmöglich ist wie es scheint. Mit etwas Bescheidenheit, der Akzeptanz des eigenen Unwissens und sehr viel Offenheit zum Lernen kann man diese Konzepte nicht nur verstehen und umsetzen, sondern auch erweitern, auf Ihnen aufbauen, vielleicht anderen Menschen helfen und auf dem Weg zu einem besseren Informatiker werden.

11. Quellen- und Literaturverzeichnis

https://github.com/Simuschlatz/CheapChess, 15.01.23, Simon Ma, CheapChess Quellcode für detaillierte Beschreibungen der angewandten Methoden
https://www.chessprogramming.org/Main_Page, 15.01.23, Chessprogramming Wiki, Wikipedia für Schachprogrammierung
https://en.wikipedia.org/wiki/Monte_Carlo_tree_search, 15.01.23, Wikipedia, Monte-Carlo-Tree-Search
https://de.wikipedia.org/wiki/Xiangqi, 15.01.21, Wikipedia, Xiangqi
https://www.xiangqi.com/, 15.01.23, Xiangqi.com, Play Chinese Chess For Free on the #1 Xiangqi Site!
https://www.deepmind.com/, 15.01.23, Deepmind, Tocherfirma von Google
https://arxiv.org/abs/1712.01815, Deepmind, Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm
https://www.youtube.com/c/SebastianLague, 15.01.23, Sebastian Lague
https://www.youtube.com/watch?v=l-hh51ncgDI, 21.08.22, Sebastian Lague, Algorithms Explained – minimax and alpha-beta pruning
https://en.wikipedia.org/wiki/Game_complexity 15.09.22, Wikipedia, Game complexity