hochschule mannheim

Entwicklung eines Efficiently Updatable Neural Network (NNUE) zur Evaluation von Schachpositionen

Marvin Karhan

Bachelor-Thesis

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

Studiengang Informatik

Fakultät für Informatik

Hochschule Mannheim

28.09.2022

Betreuer

Prof. Dr. Jörn Fischer, Hochschule Mannheim

Prof. Dr. Thomas Ihme, Hochschule Mannheim

Karhan, Marvin:
Entwicklung eines NNUE zur Evaluation von Schachpositionen / Marvin Karhan. –
Bachelor-Thesis, Mannheim: Hochschule Mannheim, 2022. 42 Seiten.

Karhan, Marvin:
Development of an NNUE for the Evaluation of Chess Positions / Marvin Karhan. –
Bachelor Thesis, Mannheim: University of Applied Sciences Mannheim, 2022. 42 pages.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 28.09.2022

Marvin Karhan

Abstract

Entwicklung eines NNUE zur Evaluation von Schachpositionen

Abstract

Development of an NNUE for the Evaluation of Chess Positions

Abstract

Inhaltsverzeichnis

Kapitel 1

Einleitung

Computerschach ist ein viel betrachtetes Thema. Schon Alan Turing und Claude Shannon haben sich damit befasst [1], [2]. In seinem 1950 verfassten Paper beschrieb Shannon [2] die Funktion zu Evaluation einer Schachposition. Ihm war jedoch auch klar, dass es wahrscheinlich niemals eine exakte Evaluation für Schach geben wird. Deshalb liegt es nahe, dafür ein Neuronales Netz (NN) zu verwenden, denn dessen Aufgabe ist es, eine solche Funktion zu approximieren. Leider ist es für die Evaluation in einem Schachcomputer wichtig, sowohl genau als auch schnell die Position zu bewerten. Je genauer die Stellung bewertet wird, desto stärker spielt das Programm. Je schneller die Bewertung stattfindet, desto weiter kann der Computer voraussehen, was ebenfalls zu einer höheren Spielstärke führt. Herkömmliche NN-Architekturen scheitern jedoch an einer zu langen Berechnungszeit oder bei sehr kleinen Netzen an einer zu ungenauen Bewertung.

Eine Lösung für die Probleme herkömmlicher NNs wurde 2018 von Nasu [3] in seinem japanischen Paper vorgestellt. Er erkannte, dass inkrementelle Aktualisierungen, wie sie bereits in Hand-Crafted Evaluation (HCE) verwendet wurden, in NNs verwendet werden können. Der Schlüssel dafür ist ein binäres und dünn besetztes Feature Set, basierend auf den Figuren und ihren Positionen. Die sogenannte Feature-Transformator-Schicht muss nicht bei jeder Aktivierung alle Elemente seines Ausgabevektors neu berechnen.

Die NNUE-Architektur ist darauf ausgelegt, schnell auf einer CPU zu laufen. Sie nutzt CPU-basierte Optimierungsmöglichkeiten wie Single Instruction, Multiple Data (SIMD) und die im letzten Absatz genannten inkrementellen Aktualisierungen, um die Geschwindigkeit zu erlangen und ihre Nutzung als Evaluationsfunktion zu rechtfertigen.

Nasu [3] hat die NNUE-Architektur für die Verwendung in der japanischen Schachvariante Shogi entwickelt. Shogi unterscheidet sich in einigen Punkten vom herkömmlichen Schach. Es hat unter anderem eine andere Spielfeldgröße und erlaubt es, geschlagene Figuren wieder einzusetzen. Trotzdem eignet sich Nasu's [3] Ansatz für traditionelles Schach, da die Zuggenerierung sowie die Evaluation ähnlich ist. Außerdem gibt es in beiden Varianten einen König, praktisch für die Auswahl eines passenden Feature Sets, wie in Unterabschnitt 2.4.1 genauer erläutert.

Nur zwei Jahre später zeigte eine Portierung des Konzepts starke Verbesserungen in dem Schachcomputer Stockfish, der sich durch NNUE um mehr als 80 Elo verbessern konnte [4], die größte Verbesserung einer Stockfish-Version jemals. Mit Ausnahme von AlphaZero [5] hatte bis dahin noch kein NN-basierter Ansatz Erfolge gezeigt.

Ein Schachcomputer besteht aus drei Teilen: Suche, Zuggenerierung (Boardrepräsentation) und Evaluation [6]. Als Basis für diese Arbeit wird ein simpler Schachcomputer, der in dem Modul Künstliche Intelligenz für autonome Systeme (KIS) entwickelt wurde, verwendet. Dieser Schachcomputer verfügt über eine simple Suche und eine HCE [7]. Gegenstand dieser Arbeit ist es, die HCE des 2021 im Modul KIS entwickelten Schachcomputers durch ein eigens trainiertes NNUE zu ersetzen. Ziel ist es hierbei nicht, eine neue NNUE-Architektur zu präsentieren. Es wird die Architektur verwendet, die von Nasu [3] vorgestellt und auch in der ersten Version der Stockfish NNUE verwendet wurde. Der Grund dafür ist, dass sie mit minimalem Domänenwissen auskommt und so ein besseres Bild der Kernelemente der NNUE-Architektur vermittelt. Außerdem sollte sie, gemessen an dem Erfolg in Stockfish, ausreichen, um die Spielstärke des in KIS entwickelten Schachcomputers zu steigern. Die NNUE-Implementierung soll ein Proof of Concept sein. Die Erstellung neuer Eingabedaten ist nicht Teil dieser Arbeit.

Kapitel 2

Grundlagen

In diesem Kapitel wird das Wissen vermittelt, welches benötigt wird, um zu verstehen, wie NNUEs im Rahmen von Schachcomputern funktionieren. Zuerst wird die Evaluation, wie sie in herkömmlichen Schachcomputern funktioniert, erklärt, auch HCE genannt. Weiterhin wird auf die grundlegenden Bestandteile, die für überwacht lernende Feedforward Neural Networks (FNNs) von Bedeutung sind, eingegangen. Außerdem wird erläutert, was SIMD ist und wie diese Vektoroperationen in C/C++ verwendet werden. Zuletzt wird die grundlegende Funktionsweise von NNUEs vermittelt, die auf den davor gelegten Grundsteinen aufbaut.

2.1 Hand-Crafted Evaluation

Es ist wichtig zu wissen, wie die HCE eines Schachcomputers funktioniert, da sie nicht nur die Variante ist, die in jedem starken Schachcomputer vor 2017 eingesetzt wurde, sondern auch heute noch in Kombination mit NNUE eingesetzt wird. Bei NNUE-Schachcomputern wird sie oft in Kombination mit der NN-Evaluation genutzt, weil sie besser in extremen Stellungen funktioniert. Besitzt beispielsweise Weiß in einer Position eine Dame mehr, muss nicht die teurere Berechnung des NNUEs durchgeführt werden, um zu entscheiden, dass Weiß im Vorteil ist.

Die HCE einer Schachposition ist eine heuristische Methode der Position einen numerischen Wert zuzuordnen. Vor der Verbreitung von NNs war HCE die einzige Form der Positions-Evaluation. Gäbe es unendliche Ressourcen, könnten aus jeder Position alle möglichen Zugfolgen per Brute Force bestimmt und den Positionen einer der drei Werte: -1 (Verlust), 0 (Remis), 1 (Gewinn) gegeben werden. In der

Realität ist es nicht möglich, den exakten Wert der Stellung zu kennen. Deshalb wird in der HCE versucht, anhand von Menschen festgelegten Kriterien der Position einen Wert zuzuordnen. Die so gewonnene Bewertung wird in der Zugsuche verwendet, um den besten Zug, abhängig von den per Hand gewählten Kriterien, zu finden. Die Evaluation wird aus Sicht der Seite, die gerade am Zug ist, angegeben. Das ist wichtig für den verwendeten Suchalgorithmus (Alpha-Beta-Suche) [8].

Die HCE eines Schachcomputers ähnelt in einigen Aspekten mehr einer Philosophie als einer Funktion. Schach ist ein Spiel, das es seit über 1000 Jahren gibt. In dieser Zeit haben Menschen Regeln überlegt, um besser Schach zu spielen. All diese Regeln in die Evaluationsfunktion zu integrieren, ist nicht ratsam. Es ist ein Abwägen zwischen Wissen und Geschwindigkeit. Je mehr Regeln dem Computer gegeben werden, umso weniger weit kann er vorausschauen.

Wenn ein Mensch Schach spielen lernt, ist der Wert der Figuren eines der ersten Erkenntnisse. Das ist ebenfalls der wichtigste Faktor für einen Schachcomputer, wie schon Shannon 1950 [2] erkannte. Die Angabe der Materialwertung wird bei Computern als Centipawn (CP) angegeben, um so mehr Spielraum für feingranulare Faktoren zu lassen. Figuren werden ebenfalls anhand ihrer Position bewertet. Dafür gibt es sogenannte Piece Square Tables, die jeder Figur abhängig von ihrer Position einen Wert zuordnen. Beispielsweise ist ein Springer am Rand des Brettes deutlich weniger wert als einer im Zentrum, auch bekannt als „ein Springer am Rand bringt Kummer und Schand“. Weitere nennenswerte Aspekte der HCE sind die Mobilität und Schwachstellen [9, S. 228].

Mobilität beschreibt die Beweglichkeit der Figuren. Sie kann aus der Anzahl der Felder, auf die eine Figur ziehen kann, berechnet werden. Das ist unbrauchbar, weil unbeschützte Felder und die Felder gefesselter Figuren nicht Teil der Mobilität sein sollten [9, S. 228]. Mit Schwachstellen sind ungeschützte Figuren, die Sicherheit des Königs, Probleme in der Bauernstruktur und Figuren, die höherwertige Figuren angreifen, gemeint [9, S. 228].

Viele der HCE-Aspekte profitieren von einer Differenzierung verschiedener Spielphasen. Schach lässt sich in drei Spielphasen teilen: die Eröffnung, das Mittelspiel und das Endspiel [9, S. 8]. Beispielsweise ist es sinnvoll, den Wert der Figuren an die aktuelle Spielphase anzupassen. Ein Bauer im Endspiel ist mehr wert als in der Eröffnung. Zwischen den verschiedenen Phasen wird meist durch die Anzahl der Figuren unterschieden. Da zwischen zwei ähnlichen Stellungen, die in zwei unter-

schiedlichen Phasen sind, kein großer Unterschied durch den Phasenwechsel entsteht, ist es sinnvoll einen Wert für beide Phasen zu berechnen und dazwischen zu interpolieren.

2.2 Neuronale Netze

Künstliche Neuronale Netze (KNNs), oder einfach NNs genannt, sind Computersysteme, die dem biologischen Vorbild des Gehirns nachempfunden sind. Analog zu seinem biologischen Vorbild besteht ein NN aus Neuronen, die miteinander vernetzt sind. Jedes Neuron reagiert auf eingehende Signale mit einer bestimmten Ausgabe. Diese Ausgabe kann sich durch neu gewonnene Erfahrungen anpassen und ermöglicht, zukünftig besser zu reagieren.

In Abbildung 2.1 ist ein einfaches Neuronales Netz zu sehen. Es besteht aus drei Schichten. Die erste Schicht, die Eingabeschicht, nimmt Eingabedaten entgegen. Eingabedaten können ganz unterschiedliche Daten repräsentieren. Ist der Eingabedatensatz beispielsweise ein 100×100 Schwarz-Weiß-Bild, ist dies eine Möglichkeit die Eingaben darzustellen. Die Eingabeschicht besteht dann aus 1000 Neuronen, die pro Neuron den Zustand eines Pixels (0 = Weiß, 1 = Schwarz) des Bildes gefüttert bekommen. Die zweite Schicht heißt versteckte Schicht, weil von außen nur die Eingabedaten und das Ergebnis sichtbar ist. Sie empfängt die Informationen der Eingabeschicht, gewichtet sie und gibt sie an die Ausgabeschicht weiter. Die versteckte Schicht kann aus mehreren Schichten bestehen. Ein NN mit mehr als drei versteckten Schichten heißt Deep Neural Network (DNN). Die letzte Sicht, die Ausgabeschicht, spiegelt das Ergebnis des NNs wider. Ein Netz, das versucht Bilder zwischen Hunden und Katzen zu unterscheiden, kann zwei Ausgabeneuronen enthalten, eins für die Wahrscheinlichkeit, dass auf dem gegebenen Bild ein Hund ist und eins für die Wahrscheinlichkeit, dass es eine Katze ist. Ein NN kann auch nur ein Ausgabeneuron besitzen, wie z. B. bei der Evaluation einer Schachposition nötig ist. Die Verbindungen der einzelnen Neuronen stellen deren Zusammenhang dar. Wie stark die Abhängigkeit ist, wird durch Gewichte definiert [10, S. 2–7].

Es gibt verschiedene Modelle neuronaler Netze. Für diese Thesis sind lediglich Feedforward Neural Networks relevant. FNNs basieren auf dem von Rosenblatt [11] beschriebenen mehrlagigen Perzeptron. Das FNN zeichnet sich durch seinen

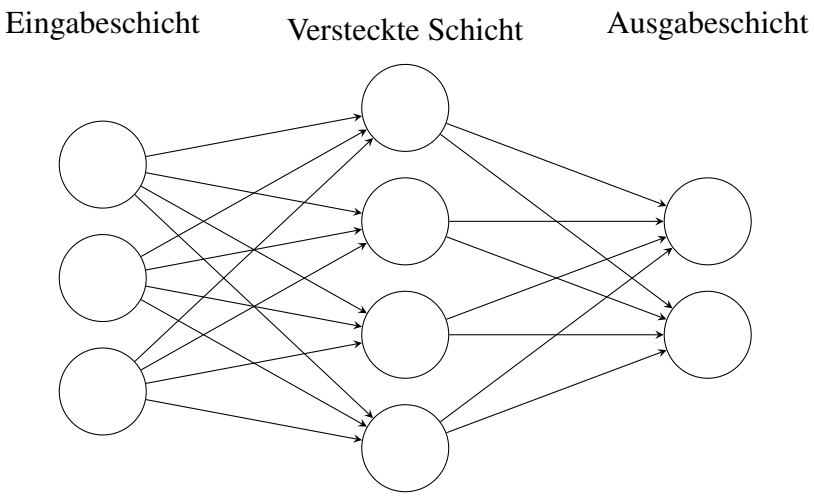


Abbildung 2.1: Ein einfaches Neuronales Netz. Jedes Neuron einer Schicht ist in Pfeilrichtung mit allen Neuronen der nächsten Schicht vernetzt. Die Verbindungen symbolisieren Gewichte. In den Neuronen werden die eingehenden Gewichte aufsummiert und über eine Transferfunktion zu einem Ausgabewert abgebildet

zyklenfreien Aufbau aus. Der Datenfluss führt immer von der Eingabeschicht zur Ausgabeschicht. Das FNN gilt als die einfachste Netzwerkarchitektur [12].

In der Praxis, so auch in dieser Arbeit, werden für die Entwicklung Neuronaler Netze Frameworks verwendet. Sie abstrahieren große Teile der Komplexität. Trotzdem ist es wichtig, ihre Funktionsweise zu kennen, um Entscheidungen treffen und Probleme beheben zu können. In den folgenden Unterabschnitten wird grundlegend auf die Einzelteile Neuronaler Netze eingegangen. Zuerst wird das Neuron beschrieben und wie sich seine Aktivität berechnen lässt. Das Unterkapitel Backpropagation beschreibt, wie NNs lernen.

2.2.1 Das Neuron

Das Neuron ist der elementare Bestandteil eines NNs. Es wurde 1943 von McCulloch und Pitts [13] eingeführt. Neuronen sind in einem NN mit anderen Neuronen verbunden und bilden so beliebig komplexe Funktionen ab. In Abbildung 2.2 ist ein einzelnes Neuron zu sehen. Die Eingänge x_0 bis x_n werden mit den Gewichten w_0 bis w_n multipliziert, aufsummiert und mit der Aktivierungsfunktion $f(\varphi)$ aktiviert. Für gewöhnlich ist immer $x_0 = 1$, was ihn zu dem Bias des Neurons mit $w_0 = b$ macht. Das bedeutet, dass es nur n tatsächliche Eingabewerte gibt: von x_1 bis x_n. Konkret lässt sich die Aktivität eines Neurons mit der Gleichung 2.1 und die Ausgabe y mit der Gleichung 2.2 bestimmen:

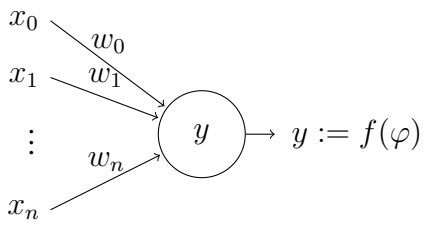


Abbildung 2.2: Ein einzelnes Neuron mit seinen Eingabe- und Ausgabekomponenten. Die Eingabewerte x werden mit den gewichten w multipliziert und im Neuron aufsummiert. Anschließend werden sie mit der Transferfunktion $f(\varphi)$ auf einen Ausgabewert abgebildet

$$f(\varphi) = \varphi(\sum_{i=0}^{n} w_i x_i) \tag{2.1}$$

$$y = f(\varphi) \tag{2.2}$$

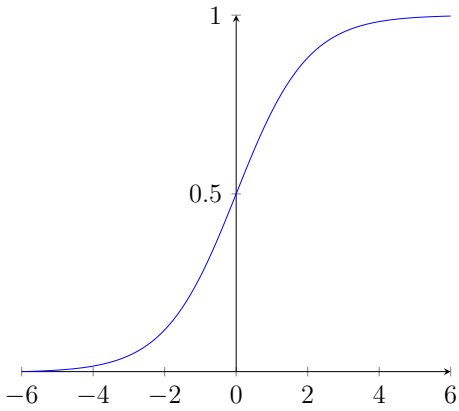
Die Aktivierungsfunktion, oder auch Transferfunktion, eines Neurons kann linear oder nicht linear sein. Ist die Transferfunktion linear, ergibt ein mehrschichtiges NN keinen Sinn, da sie zu einer Schicht vereinfacht werden kann. Außerdem sind lineare NNs nicht in der Lage, nicht lineare Probleme zu lösen [14]. Nicht lineare Transferfunktionen sind interessanter, da sie für nicht lineare Probleme Antworten liefern.

In Abbildung 2.3 sind zwei Aktivierungsfunktionen zu sehen. In Abbildung 2.3a ist eine Standardsigmoide abgebildet. Sie sorgt dafür, dass die Ausgabe des Neurons immer zwischen null und eins ist. Berechnet wird sie mit Gleichung 2.3. Abbildung 2.3b zeigt eine Rectified Linear Unit (ReLU)-Transferfunktion. Der niedrigste Wert ist mindestens null. Konkret ist die Berechnung in Gleichung 2.4 angegeben. ReLU geht bis ins Unendliche. Wegen der aggressiven Quantisierung wird der Bereich der Aktivierungsfunktionen ebenfalls nach oben begrenzt, auch ClippedReLU (siehe Gleichung 2.5) genannt [15].

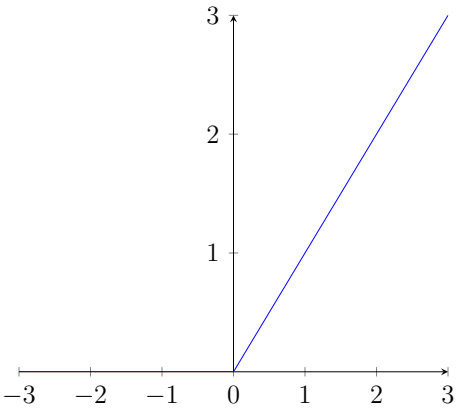
$$Sigmoid(x) = \frac{1}{(1 + e^{-x})} \tag{2.3}$$

$$ReLU(x) = max(0, x) \tag{2.4}$$

$$ClippedReLU(x) = min(max(0, x), 1) \tag{2.5}$$



(a) Standardsigmoide Aktivierungsfunktion. Sie ist Konvex für Werte unter einem bestimmten Punkt und Konkav für Werte darüber. Hier ist dieser Punkt 0. Außerdem nähern sich die Extreme der Funktion den Grenzwerten 0 und 1



(b) Rectified linear Aktivierungsfunktion. Die Ausgabe der Funktion beträgt immer mindestens den Wert null und ist sonst linear

Abbildung 2.3: Beispiele für Aktivierungsfunktionen

2.2.2 Backpropagation und Gradientenabstieg

Das Besondere an NNs ist, dass sie lernen, also nach und nach besser werden. Für gewöhnlich wird erwartet, dass ein Programm bei selber Eingabe immer dasselbe Ergebnis liefert. Ein NN hingegen lernt während seiner Trainingsphase aus Fehlern. Es gilt, den Fehler zu minimieren. Dabei handelt es sich um ein Optimierungsproblem. Als Lösung dafür wird der Gradientenabstieg verwendet. Es gibt dafür auch andere Methoden, auf die hier nicht weiter eingegangen wird [15].

Der Gradientenabstieg minimiert den Fehler, indem er dem negativen Gradienten einer Verlustfunktion folgt [16]. Die Annahme ist, dass die Richtung des Gradienten einer Funktion diese maximiert. Deshalb führt ein Schritt in die entgegengesetzte Richtung zu einer Minimierung der Funktion, also einer Minimierung des Fehlers. Die Lernrate steuert die Schrittweite, die dem negativen Gradienten folgt. Sind die Schritte zu groß, wird das (lokale) Minimum übersprungen. Sind sie klein, dauert die Konvertierung länger oder bleibt in einem lokalen Minimum hängen. Um beide dieser Probleme bestmöglich zu vermeiden, ist die Lernrate nicht konstant, sondern ändert sich über den Trainingszeitraum.

Die Lernrate kann entweder abhängig durch einen festen Plan oder durch ein adaptives Modell angepasst werden. Ein Beispiel für eine fest geplante Änderung ist die Multiplikation der Lernrate mit einem Faktor x alle n Epochen. Eine Epoche ist normalerweise ein Durchgang des gesamten Eingabedatensatzes. Alternativ gibt es verschiedene adaptive Lernraten-Modelle. In dieser Arbeit wird z. B. Adadelta

verwendet. Adadelta ist eine Adaption des Gradientenabstiegs. Diese Variante erweitert den Gradientenabstieg um eine dynamische Lernrate, die akkumulierende Gradienten durch ein Fenster löst. In dem Fenster wird nur die Summe der letzten Gradienten, bestimmt durch eine feste Größe, akkumuliert [17]. Adadelta zeigt gute Leistungen im Vergleich zu anderen adaptiven Lernraten-Modellen und eliminiert das Problem, eine passende Lernrate zu finden.

Es gibt drei Varianten des Gradientenabstiegs [16]:

- *Batch Gradientenabstieg*, berechnet den Gradienten der gesamten Verlustfunktion über den gesamten Trainingsdatensatz.
- *Stochastischer Gradientenabstieg*, berechnet den Gradienten für jedes Trainingsbeispiel einzeln.
- *Mini-batch Gradientenabstieg*, berechnet den Gradienten für jedes Subset der Größe n der Trainingsbeispiele.

Die vorherigen Absätze erklären, wie der Fehler durch einen Gradientenabstieg minimiert werden kann. In Unterabschnitt 2.2.1 wird gezeigt, wie die Eingaben und Gewichte eines Neurons zu einem Ergebnis führen. Nun stellt sich die Frage: Welche Werte müssen die Gewichte haben, um den Fehler zu minimieren? Die perfekten Gewichte eines NNs lassen sich nicht berechnen. Dafür ist die Anzahl der Faktoren zu groß.

Als Backpropagation wird das Verfahren der Fehlerrückführung beschrieben. Es gehört zu der Familie der überwachten Lernverfahren. Damit wird der negative Gradient der Verlustfunktion rückwärts durch das Netz geführt. Dabei werden die Werte der Gewichte angepasst [18]. Die Gewichte werden mithilfe der rekursiven Anwendung der Kettenregel aus der Infinitesimalrechnung und Berechnung einer Ableitung der Unterfunktion einer bekannten übergeordneten Funktion angepasst.

2.2.3 Verlustfunktion

Da ein NN aus seinen Fehlern lernen kann, muss ermittelt werden, ob das Netz mit seiner Vorhersage richtig liegt. Mit der Verlustfunktion wird der Fehler an einem bestimmten Punkt der zu bestimmenden Funktion ermittelt. Dafür wird ein Satz von Eingabedaten genommen und dem Netz gegeben, welches eine Vorhersage für eine Ausgabe abhängig von den Gewichten trifft. Diese Ausgabe wird mit dem vorher

definierten zu erwartenden Ergebnis verglichen. Basierend darauf wird der Gradient der Verlustfunktion berechnet, der von den in Unterabschnitt 2.2.2 beschriebenen Methoden zur Anpassung der Gewichte verwendet wird. In folgendem Text sind zwei Verlustfunktionen beschrieben, die häufig in NNs eingesetzt werden.

Die mittlere quadratische Fehler-Verlustfunktion ist eine simple Verlustfunktion. Sie nimmt die Summe der quadratischen Differenz der vorhergesagten Werte y mit den Zielwerten t über eine Menge von n Eingabewerten. Das Ergebnis ist eine quadratische Funktion, dessen Gradienten sich gut für Gradientenabstieg eignet. Die entsprechende Funktion ist in Gleichung 2.6 zu sehen.

$$MSE(y,t) = \frac{1}{n} * \sum_{i=1}^{n} (y_i - ti)^2 \tag{2.6}$$

Die Kreuzentropie-Verlustfunktion eignet sich für Klassifizierungsprobleme [19]. Die Evaluation einer Schachposition kann als ein solches Problem behandelt werden [15]. Sie setzt sich aus der Summe der tatsächlichen Wahrscheinlichkeit p und dem Logarithmus der vorhergesagten Wahrscheinlichkeit q über alle Klassen X der Verteilung zusammen. Für das Beispiel einer Schachevaluation wird statt der Wahrscheinlichkeit einer bestimmten Klasse die Sigmoide der CP Evaluation genommen [15]. Die Stellung wird also anhand der Wahrscheinlichkeit auf Sieg/Remis/Verlust klassifiziert. Konkret lässt sich die Kreuzentropie mit der Gleichung 2.7 berechnen.

$$H(p|q) = -\sum_{x \in X} p(x) * log(q(x)) \tag{2.7}$$

2.2.4 Quantisierung

Quantisierung ist ein Signalverarbeitungsverfahren, bei welchem Eingabewerte auf eine vorher festgelegte kleinere Menge von Ausgabewerten abgebildet werden. Ein simples Beispiel für Quantisierung ist das Abbilden von rationalen Zahlen auf ganze Zahlen. Hierfür werden die rationalen Zahlen zu der nächsten ganzen Zahl gerundet. Im Bereich der Informatik werden für Gleitkomma-Eingabewerte oft Festkommazahlen oder Ganzzahlen als Ausgabewerte gewählt [20]. Egal wie die Quantisierung stattfindet, das Ziel ist es, weniger Speicherkapazität und weniger Berechnungszeit zu benötigen mit minimalem Präzisionsverlust. Welches Quantisierungsschema ver-

wendet wird, hängt von dem Anwendungsfall ab und kann nicht allgemein bestimmt werden. Es ist immer ein Abwägen von Leistung und Präzision.

Dieses Verfahren eignet sich gut für Anwendungsgebiete mit wenig Speicher- und Rechenkapazität, wie beispielsweise der Einsatz von NNs bei Mobilgeräten [20], [21]. Der Grund dafür ist zweierlei. Erstens sorgt Quantisierung dafür, dass weniger Platz im Cache der CPU gebraucht wird, wodurch weniger Schreib- und Lesezugriffe ausgeführt werden und somit die Berechnung schneller ist. Zweitens ermöglicht die Abbildung auf kleinere Datentypen einen Performance-Gewinn durch die effizientere Verwendung von prozessorinternen Recheneinheiten, die beispielsweise SIMD unterstützen. Zudem ermöglicht die Abbildung auf Ganzzahl-Typen die Nutzung von CPU-internen Ganzzahl-Recheneinheiten, die effizienter als die Gleitkommazahl-Äquivalente funktionieren, falls überhaupt vorhanden [22].

Das Problem der Quantisierung ist das Einbauen von „Fehlern“. Bei NNs wird oft von Fehler-Kumulierung gesprochen, da bei der Aktivierung eines NNs in jedem quantisierten Neuron der Fehler wächst [23].

2.3 SIMD

In diesem Abschnitt geht es um SIMD. SIMD ermöglicht Prozessor-Anweisungen, die eine Instruktion auf mehrere Elemente eines Vektors gleichzeitig durchführen. Es gibt je nach Mikroprozessor-Architektur verschiedene Erweiterungen, um SIMD zu implementieren. In dieser Arbeit sind alle Beispiele mit dem Advanced Vector Extensions 2 (AVX2)-Befehlssatz, welcher auf Advanced Vector Extensions (AVX) aufbaut, beschrieben. Der Grund dafür ist, dass AVX2 von modernen Intel- und AMD-Mikroprozessoren unterstützt werden [24, S. 117].

Der Begriff SIMD kommt von der flynnschen Klassifikation, die Rechnerarchitekturen in vier Gebiete aufteilt [25]. Die Aufteilung orientiert sich an der Anzahl vorhandener Befehls- und Datenströme. Es gibt Single und Multiple Instructions, sowie Single und Multiple Data. Die daraus entstehenden Klassen heißen: SIMD, Single Instruction, Single Data (SISD), Multiple Instruction, Multiple Data (MIMD) und Multiple Instruction, Single Data (MISD).

SIMD kann über drei Wege realisiert werden. Auf der tiefsten Ebene in Assemblersprache hat der Programmierer die Verantwortung, die Vektorisierung, die Registerzuweisung und das Befehlsscheduling. Das Problem hierbei ist leider, dass

Menschen nicht perfekt sind und der darin bessere Compiler Teile dieser Aufgaben übernehmen kann. Dafür gibt es intrinsische Funktionen, die in Programmiersprachen wie C/C++ zur Verfügung stehen. Sie kapseln prozessorspezifische Operationen in Funktionsaufrufe. Für den AVX2-Befehlssatz gibt es in C/C++ das `immintrin.h` Header-File. Mit der Verwendung von intrinsischen Funktionen übernimmt der Programmierer nur die Vektorisierung des Codes, die Registerzuweisung und das Befehlsscheduling werden vom Compiler übernommen. Die dritte Möglichkeit ist die automatische Vektorisierung. Dabei übernimmt der Compiler alle Aufgaben. Die Limitierungen für den Compiler sind dabei groß [26]. Der Compiler kann nicht sicherstellen, dass die zu vektorisierenden Daten in einem zusammenhängenden Speicherbereich oder entsprechend aligned sind [24, S. 118-120]. Die beste Variante ist die Verwendung der intrinsischen Funktionen, die ein Maximum an Flexibilität und Compiler-Optimierungen bietet.

In den folgenden Unterkapiteln wird erläutert, warum Memory Alignment wichtig ist und wie die für NNUE wichtigen intrinsischen Funktionen in C/C++ funktionieren.

2.3.1 Registerverwaltung

Um SIMD-Befehle effizient auszuführen, werden die Variablen, in anweisungspezifischen Registern abgelegt, der Registergröße entsprechend ausgerichtet, auch Alignment genannt. Dafür gibt es zwei Möglichkeiten, entweder die Werte sind bei der Definition bereits aligned oder sie werden beim Laden in das Register aligned. Für beide Varianten gibt es Anweisungen [27]. Die Anweisungen für unaligned Variablen, aligned die Daten zuerst und sind deshalb deutlich langsamer. Das Alignment zur Definition ist deshalb präferiert und kann in C++ sehr einfach über den Spezifizierer *alignas* implementiert werden. *Alignas* nimmt einen Integer, der das geforderte Alignment in Byte spezifiziert. Im Quellcodeverzeichnis 2.1 wird ein Array 32 Byte aligned, passend für ein AVX2-Register.

```
alignas(32) int aligned[64];
```

Quellcodeverzeichnis 2.1: 32 Byte Aligned Array

Je nach Befehlssatz gibt es unterschiedlich große Register. Die in AVX2 verwendeten Register heißen *ymm* und haben eine Registerbreite von 256 Bit [27]. Bei der Unterstützung mehrerer Befehlssätze mit unterschiedlich großen Registern ist

es empfehlenswert, die Variablen auf die größtmögliche Registergröße anzupassen. Der Schachcomputer, der im Rahmen dieser Arbeit verwendet wird, unterstützt bis zu Advanced Vector Extensions 512 (AVX512), also 64-Byte-Registerbreite.

AVX2-Befehle erwarten die Vektoren in einem *ymm*-Register. Deshalb wird der Eingabevektor zuerst zu dem Typen `__m256i` konvertiert und in ein Register geladen. Die Information, um welche Daten es sich handelt, geht dabei verloren. Für unterschiedliche Integer-Typen gibt es jeweils einen eigenen Befehl. Die Aufgabe des Entwicklers ist es, die Form seiner Daten zu kennen und die entsprechenden Operationen darauf auszuführen. Spezifische Beispiele enthält der Unterabschnitt 2.3.2.

Bei Integern kann es passieren, dass es zu einem Überlauf kommt. Wird ein 8-Bit-Integer mit dem Wert 127 um eins erhöht, läuft sie über und enthält statt 128 den Wert -128. In den meisten SIMD-Befehlssätzen gibt es zu den Befehlen, die überlaufen, zusätzliche Befehle, die nicht überlaufen [27]. Die sogenannte Saturation deckelt Werte am Minimum/Maximum. In dem gerade genannten Beispiel ist das saturated Ergebnis 127.

2.3.2 Intrinsische Funktionen für NNUE

Allein AVX2 besitzt über 200 verschiedene Befehle [27]. Deshalb sind in diesem Unterkapitel die wichtigsten für eine SIMD-Implementierung für NNUE angegeben. In Tabelle 2.1 sind die wichtigen Befehle aufgelistet. Sie werden in nachfolgenden Absätzen genauer eingeordnet. Konkrete Implementierungen gibt es in Abschnitt 4.3.

Bevor eine AVX2-Operation ausgeführt werden kann, muss der zu verarbeitende Vektor in ein *ymm*-Register geladen werden. Dafür gibt es die Operation VMOVDQA. Wichtig ist, dass der Vektor 32-Byte aligned ist. Diese Operation gibt es auch für unaligned Vektoren (VMOVDQU), ist aber für diese Arbeit nicht relevant. Dieselbe Operation (VMOVDQA) ist zuständig für das Speichern der Vektoren eines *ymm*-Registers in einen herkömmlichen Vektor. Die intrinsische Funktion ist jedoch eine andere.

Die SIMD-Operationen, welche in der NNUE-Evaluation gebraucht werden, lassen sich in drei Teile untergliedern. Erstens die ClippedReLU, welches die Transferfunktion aller verwendeten Schichten ist und sich lediglich in der Größe der Integer der verschiedenen Schichten unterscheidet, abhängig von dem Quantisierungssche-

Tabelle 2.1: Liste der für NNUE wichtigen AVX2-Befehle. In der Liste enthalten ist der Name des Befehls, die intrinsische Methodensignatur und eine kurze Beschreibung [27].

Befehl	Funktion	Beschreibung
VMOVDQA	`__m256i _mm256_load_si256 ( __m256i const * mem_addr)`	Lädt 256-Aligned-Bits aus *mem_addr* in ein *ymm* Register.
VMOVDQA	`void _mm256_store_si256 ( __m256i * mem_addr, __m256i a)`	Speichert 256-Bits aus einem *ymm* Register in einen 32-Byte-Aligned Vektor (*mem_addr*).
VPXOR	`__m256i _mm256_setzero_si256 ( void)`	Gibt ein Vektor des Typen `__m256i` mit nur Nullen zurück.
VPACKSSWB	`__m256i _mm256_packs_epi16 ( __m256i a, __m256i b)`	Nimmt die zwei 16-Bit Vektoren *a* und *b* und packt sie in einen 8-Bit Vektor mithilfe von Saturation.
VPMAXSB	`__m256i _mm256_max_epi8 ( __m256i a, __m256i b)`	Vergleicht die zwei 8-Bit Vektoren *a* und *b* und speichert die jeweils größte Zahl.
VPERMQ	`__m256i _mm256_permute4x64_epi64 (__m256i a, const int imm8)`	Ordnet 64-Bit große Integer anhand von der Maske *imm8* an.
VPADDW	`__m256i _mm256_add_epi16 ( __m256i a, __m256i b)`	Addiert 16-Bit Integer der Vektoren *a* und *b*.
VPSUBW	`__m256i _mm256_sub_epi16 ( __m256i a, __m256i b)`	Subtrahiert 16-Bit Integer der Vektoren *a* und *b*.
VPSRAW	`__m256i _mm256_srai_epi16 ( __m256i a, int imm8)`	Schiebt die 16-Bit Integer des Vektors *a* um *imm8* nach links.
VPMADDUSBW	`__m256i _mm256_maddubs_epi16 ( __m256i a, __m256i b)`	Multipliziert die vorzeichenlosen 8-Bit Integer des Vektors *a* mit den 8-Bit Integern in *b* und speichert das Ergebnis in einem 16-Bit Integer zwischen. Danach werden benachbarte Integer in saturated 16-Bit Integer gespeichert.

ma. Zweitens der Akkumulator, der die Eingabewerte transformiert (siehe Unterabschnitt 2.4.2). Drittens die affine Transformation der linearen Schichten bzw. das Matrixprodukt, also Gewichtsmatrix multipliziert mit dem Eingabevektor der jeweiligen Schicht.

Für die Clipped ReLU Transferfunktion sind die vier Befehle relevant: VPXOR, VPACKSSWB, VPMAXSB, VPERMQ. Die nächsten zwei aufgelisteten Befehle in der Tabelle 2.1 (VPADDW, VPSUBW) sind Teil des Akkumulators. VPSRAW und VPMADDUSBW sind Befehle, die in der affinen Transformation verwendet werden. VPSRAW ist nicht zwingend nötig, wird aber aufgrund des gewählten Quantisierungsschemas benötigt.

2.4 NNUE

Die NNUE-Evaluationsfunktion evaluiert eine Schachposition auf einer CPU ohne eine Notwendigkeit für eine GPU. Damit die NNUE-Evaluation eine Chance hat, besser als die HCE zu sein, muss sie schnell berechenbar sein. Anderenfalls geht sie nicht weit genug in die Zukunft. Eine Untersuchung der Relation zwischen Suchtiefe und Spielstärke des Schachcomputers Houdini 2013 hat ergeben, dass die Suchtiefe einen sehr großen Einfluss auf die Spielstärke hat, aber auch, dass dieser Effekt mit zunehmender Tiefe kleiner wird [28]. Ein weiterer Vorteil von NNUEs ist, dass sie ein Eins-zu-eins-Ersatz für HCEs sind. Es wird lediglich ein Netz und der CPU-optimierte Code zur Verwendung des Netzes benötigt.

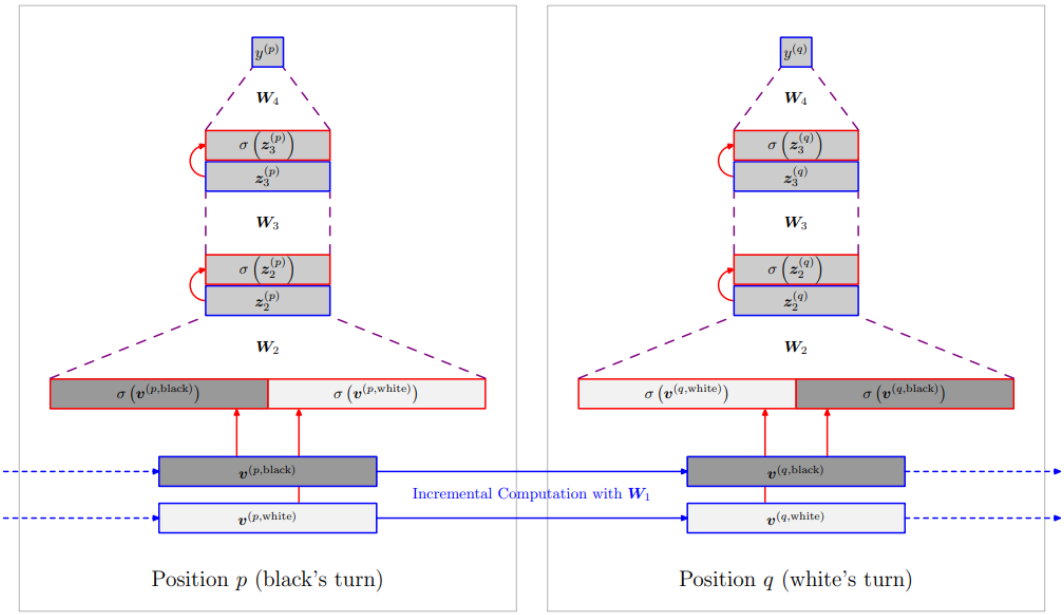


Abbildung 2.4: NNUE-Evaluationsfunktion für die Evaluation von Position q, dabei unterscheidet sich q von p nur um einen Zug. Abbildung für die Evaluation des Shogicomputers „the end of genesis T.N.K.evolution turbo type D“ [3]

In Abbildung 2.4 wird der Aufbau der NNUE-Evaluationsfunktion, wie er von Nasu [3] entwickelt wurde, gezeigt. Er eignet sich für die schnelle Berechnung auf einer CPU. In den folgenden Unterkapiteln wird genauer darauf eingegangen, warum das der Fall ist.

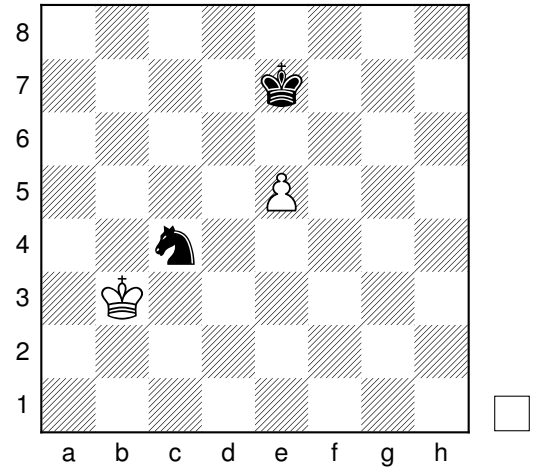


Abbildung 2.5: Exemplarische Schachposition. Weiß am Zug

2.4.1 Feature Set

Das Feature Set bestimmt die Form des Vektors, die der Eingabeschicht des NNs gegeben wird. Ein simples Feature Set setzt sich aus der Position, dem Figurentyp und seiner Farbe zusammen. Mit 64 Feldern, sechs verschiedenen Figurentypen und zwei Farben gibt es $64 * 6 * 2 = 768$ Merkmale. Ein Merkmal ist entweder 0 oder 1, je nachdem, ob auf dem Feld eine Figur mit der entsprechenden Farbe steht. Da im Schach maximal 32 Figuren im Spiel sind, kann es nur 32 gleichzeitig aktive Merkmale geben. In Abbildung 2.5 gibt es vier aktive Features: (B3, König, Weiß), (C4, Springer, Schwarz), (E5, Bauer, Weiß), (E7, König, Schwarz). Wenn der weiße König den Springer schlägt, ändern sich drei Features. Die Features (B3, König, Weiß) sowie (C4, Springer, Schwarz) werden inaktiv und ein neues Feature (C4, König, Weiß) wird aktiv. Bei diesem Feature Set ändern sich von einer Position p zu einer Position q vier Features (Rochade) maximal und im Durchschnitt drei Features [15]. Das Feature Set erfüllt die zwei Voraussetzungen, die für ein NNUE gelten:

1. Die Anzahl der aktiven Merkmale ist klein.
2. Die Anzahl der unterschiedlichen Merkmale von Position p nach Position q ist minimal.

Anhand dieser zwei Regeln lässt sich auch die Frage, warum sind nicht Elemente, die schon in der HCE verwendet werden (wie z. B. Rochade-Rechte) Teil des Feature Sets, beantworten. Es erhöht die Anzahl aktiver Merkmale und die Anzahl

durchschnittlicher Änderungen. Der Gewinn an Evaluationsgenauigkeit rechtfertigt nicht die Geschwindigkeitseinbußen [15].

Ein in der Praxis besser geeignetes Feature Set als das eingangs erklärte Beispiel ist das weit verbreitete HalfKP-Feature Set [3], [15]. Es besteht aus dem Tupel (Feld des eigenen Königs, Feld der Figur, Figurentyp, Farbe der Figur), wobei der Figurentyp kein König sein kann. Die Anzahl aktiver Merkmale ist hier maximal 30, da die Könige nicht mit enthalten sind. Die gesamte Anzahl der Merkmale ist $64*64*5*2 = 40960$. Von einer zu der anderen Position ändert sich im Schnitt öfter etwas, da bei einem Zug des Königs alle aktiven Merkmale geändert werden. Das ist eine bessere Aufteilung der Merkmale, da sich im Schach der König selten bewegt und durch dieses Feature Set das NN besser versteht, wie die Figuren in Relation zum König stehen [15]. Es ist bekannt, dass überparametrisierte Netze, also Netze mit mehr Parametern als theoretisch nötig, besser lernen und gut generalisieren. Normalerweise sorgen mehr versteckte Schichten oder größere versteckte Schichten für die Überparametrisierung [29], [30]. In dem Fall der Schachevaluation ist das aufgrund der nötigen Geschwindigkeit nicht möglich.

HalfKP allein spiegelt nicht die gesamte Position wider. Wie der Name impliziert, fehlt der gegnerische König. Deshalb werden die zwei Seiten separat behandelt. Es gibt einen Vektor pro Seite. Das bedeutet, es gibt doppelt so viele aktive Merkmale und doppelt so viele Änderungen. Insgesamt zahlt sich der Kompromiss immer noch aus [15]. Wie die zwei Vektoren kombiniert werden, ist im nächsten Unterkapitel erläutert.

HalfKP stammt aus der Shogi-Welt, in der es keine Rochade gibt und somit die Relation der Figuren zum König wichtiger ist. Für Schach gibt es keine logische Begründung, warum HalfKP eine gute Repräsentation ist. HalfKP ist nur empirisch zu rechtfertigen und bildet die Grundlage für alle anderen verwendeten Feature Sets [15].

2.4.2 Akkumulator

Wie in Unterabschnitt 2.4.1 angesprochen und in Abbildung 2.4 zu sehen, werden für die Darstellung einer Schachposition mit HalfKP zwei Vektoren benötigt. Ein Vektor $v^{(p,white)}$ für Weiß und $v^{(p,black)}$ für Schwarz. Die zwei Vektoren werden kombiniert, um sie in die nächste Schicht weiterzugeben. Für gewöhnlich geben

Schachcomputer die Evaluation immer aus der Sicht der Seite an, die gerade am Zug ist. Deshalb wird $v^{(p,white)}$ mit $v^{(p,black)}$, wenn Weiß am Zug ist und $v^{(q,black)}$ mit $v^{(q,white)}$, wenn Schwarz im nächsten Zug dran ist konkateniert. Das ist nicht zwingend notwendig, die Bewertung der Position kann ebenfalls nicht relativ zu der aktiven Seite sein. Eine Relative Sichtweise ermöglicht dem Netz ein Verständnis für Tempo ohne Performance Einbuße. Da die Eingabewerte transformiert werden, wird dieser Schritt auch Feature Transformer genannt.

Es gibt verschiedene Wege, wie die zwei Eingabevektoren gehandhabt werden [15]. Entweder beide Seiten verwenden dieselben Gewichte oder die Gewichte sind seitenspezifisch. Für den ersten Ansatz wird das Brett für Schwarz (kann auch Weiß sein) gespiegelt, weil ein weißer König auf E1 anders als ein schwarzer König auf E1 zu deuten ist. Alternativ sind die Gewichte seitenspezifisch. Dieser Ansatz scheint logischer, da Weiß und Schwarz nicht gleich spielen. Die Nachteile sind ein größeres NN und eine längere Trainingszeit.

Bisher wurde besprochen, dass der dünn besetzten Vektor des HalfKP-Feature Sets für Performanz gewinne ausnutzt wird. Wie das funktioniert, lässt sich am besten durch eine Betrachtung des Matrixprodukts der Gewichtsmatrix mit dem Eingabevektor zeigen. Zur Veranschaulichung wird die Berechnung nur für eine Seite betrachtet. Der Bias wird ebenfalls nicht beachtet, da er nur einer simplen Addition bedarf, die nicht wichtig für den Zusammenhang ist. Angenommen w_i^j ist das Gewicht für den Eingabewert i mit dem Neuronen j der ersten versteckten Schicht und x_i die Eingabe für den Eingabewert i, ergibt sich die Gleichung 2.8 für das Matrixprodukt der aktuellen Stellung $v^{(p)}$:

$$v^{(p)} = \begin{bmatrix} w_0^0 & w_1^0 & \cdots & w_{40959}^0 \\ w_0^1 & w_1^1 & \cdots & w_{40959}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_0^{255} & w_1^{255} & \cdots & w_{40959}^{255} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{40959} \end{bmatrix} \tag{2.8}$$

Fakt ist, dass ein Großteil der x_i Eingabewerte 0 ist, deshalb lässt sich die Gleichung 2.8 deutlich vereinfachen. Angenommen nur ein Eingabewert ist 1 und der Rest 0, lässt sich $v^{(p)}$ mit Gleichung 2.9 und allgemein mit Gleichung 2.10 berechnen.

$$v^{(p)} = \begin{bmatrix} w_i^0 \\ w_i^1 \\ \vdots \\ w_i^{255} \end{bmatrix} x_i \tag{2.9}$$

$$v^{(p)} = \sum_{i \in \{k | x_k \neq 0\}} \begin{bmatrix} w_i^0 \\ w_i^1 \\ \vdots \\ w_i^{255} \end{bmatrix} x_i \tag{2.10}$$

Da x_i im Fall von Gleichung 2.10 x_i immer 1 ist, kann es weggelassen werden und w_i^0 bis w_i^{255} kann mit W zusammengefasst werden:

$$v^{(p)} = \sum_{i \in \{k | x_k \neq 0\}} W(:, i) \tag{2.11}$$

Gleichung 2.11 beschreibt einen Refresh des Akkumulators, der initial und bei einem Zug des Königs durchgeführt wird. Bei einem regulären Zug wird lediglich der gespeicherte Vektor $v^{(p)}$ aktualisiert. Die Eingabewerte x_i, die sich geändert haben, addiert bzw. subtrahiert von $v^{(p)}$ ergibt das den Vektor $v^{(q)}$, der den Akkumulator der nächsten Position symbolisiert. Dies wird konkret durch die folgende Gleichung 2.12 bestimmt:

$$\begin{aligned} v^{(q)} = v^{(p)} &- \sum_{i \in \left\{k | x_k^{(p)} = 1 \wedge x_k^{(q)} = 0\right\}} W(:, i) \\ &+ \sum_{i \in \left\{k | x_k^{(p)} = 0 \wedge x_k^{(q)} = 1\right\}} W(:, i) \end{aligned} \tag{2.12}$$

Wird diese Berechnung für beide Seiten durchgeführt, der Bias addiert, die Vektoren konkateniert und mit einer Aktivierungsfunktion aktiviert, ist das Ergebnis, der Vektor, der an die erste versteckte Schicht weitergegeben wird.

Normalerweise werden Gewichtsmatrizen Reihe für Reihe im sequentiellen Speicher abgelegt. Im Fall des Akkumulators wäre das ein Nachteil, da wie beschrieben immer Spalten der Gewichtsmatrix addiert/subtrahiert werden. Deshalb wird

die Gewichtsmatrix vor dem Speichern transponiert, auch Column-Major Order genannt. Das ermöglicht SIMD-Anweisungen leichter zu nutzen, da sie auf zusammenhängenden Speicherzugriffen basieren.

Kapitel 3

Verwandte Arbeiten

Auch wenn NNUEs erst seit 2020 in Schachcomputern existieren, haben sie einen großen Einfluss auf die Schachcomputerlandschaft. In der letzten Saison (Saison 22) der Top Chess Engine Championship (TCEC) [31] spielen fünf der acht Teilnehmer der höchsten Division mit einer hybriden NNUE-Evaluation, die restlichen drei nutzen einen von AlphaZero etablierten NN Ansatz, der später in diesem Kapitel genauer erläutert ist.

Stockfish war der erste Schachcomputer mit NNUE und manifestierte so seine Stellung als stärkster Schachcomputer. Die Entwicklung ist ein Community-Projekt. Das auf SETI@home [32] basierende Testing Framework Fishtest ermöglicht das Testen tausender Versionen. Alle Änderungen der Codebasis und neue NNUEs werden durch die Plattform getestet. Stand August 2022 gibt es 289 Entwickler und 1747 Tester, die 126.000 Tests seit der Entstehung der Plattform 2013 durchgeführt haben [33]. Dieser Ansatz ist ein großer Faktor dafür, wie Stockfish der beste Schachcomputer wurde und auch zukünftig bleibt. Die meisten anderen NNUE-Schachcomputer bauen auf der Variante von Stockfish auf.

Die Architektur der Stockfish NNUE ist aktuell in seiner fünften Version. Sie besteht aus einem Feature Set mit 45.056 Eingabeparametern, namens HalfKAv2_hm, die inkrementell in zwei Farben abhängigen Akkumulatoren aktualisiert werden. Die Ausgabe der Eingabeschicht besteht aus jeweils 520 Ausgabewerten, welche in einen Vektor mit acht Werten und einen mit den restlichen 512 Werten geteilt wird. Die zwei Vektoren mit acht Werten werden basierend auf der Seite angepasst, welche am Zug ist. Anhand der Phase des Spiels wird einer der sogenannten Buckets gewählt. Konkret bestimmt die Anzahl der im Spiel stehenden Figuren die Spielphase: $\lfloor \frac{pieceCount-1}{4} \rfloor$. Dasselbe Verfahren wird auch zur Auswahl der Schichten der

versteckten Schichten verwendet. Die Buckets beinhalten zwei Schichten, die 1024 Werte gewichten und mit einer Clipped ReLU oder der Quadratwurzel einer Clipped ReLU aktivieren. Alle Schichten sind linear und die Quantisierung wird schon während des Trainings angewandt [15].

Der Unterschied von HalfKAv2_hm zu dem in dieser Arbeit verwendeten HalfKP-Feature Set ist, dass der König selbst als Figur enthalten ist. Jedoch werden die Könige egal welcher Farbe als ein Figurentyp angesehen, da die Belegung ihrer Felder disjunkt ist. Sie stehen also nie auf demselben Feld. So werden acht Prozent der Eingabeparameter gespart. „hm“ steht für „horizontally mirrored“, auf Deutsch horizontal gespiegelt. Das bedeutet, das Brett wird vor der Erstellung der Eingabeparameter gespiegelt, sodass der eigene König immer auf einem der e bis h (je nach Konvention auch a bis d) Ränge ist. Das hört sich unintuitiv an und spiegelt nicht die Realität wider, eignet sich trotzdem, da es die Größe des Netzes stark reduziert, aber ein Unterschied in Spielstärke kaum messbar ist [15].

Schon vor der Entwicklung von NNUEs gab es einen NN-basierten Schachcomputer namens AlphaZero [5], der den schon damals HCE-basierten Schachcomputer Stockfish vernichtend schlagen konnte. AlphaZero wurde 2017 von zu Google gehörendem Forschungsunternehmen DeepMind entwickelt. Es erlernte laut DeepMind bereits nach vier Stunden Self-Play reinforcement Training die nötige Spielstärke, um gegen Stockfish zu gewinnen. DeepMind nennt den Trainingsansatz *tabula rasa*. Nennenswert ist, dass zur Suche Monte Carlo Tree Search (MCTS) anstatt der normalerweise genutzten Alpha-Beta-Suche verwendet wird. AlphaZero kennt nur die Spielregeln und trainiert sein Convolutional Neural Network (CNN) durch Self-Play reinforcement Training. Der Ansatz ist generell anwendbar und hat in den Spielen Schach, Shogi und Go gegen die führenden Computerprogramme gewonnen.

Dem erstmals durch einen Schachcomputer geschlagenen damaligen Schachweltmeister Kasparov [34] gefällt der dynamische und offene Spielstil von AlphaZero, der anders als der von HCE-basierten Schachcomputern auf konventionellem Wissen aufbauende Spielstil. Er beschrieb AlphaZero als Experten und nicht als das Werkzeug eines Experten. Damit deutet Kasparov darauf hin, dass ein Schachspieler, besonders Super-Großmeister, diesen Schachcomputer nicht nur für die Analyse ihrer Züge nehmen kann, sondern auch für das Entdecken neuer Spielweisen, die vorher nicht in Betracht gezogen wurden.

Leider ist AlphaZero nicht öffentlich zugänglich und wird auch von DeepMind nicht weiter entwickelt. Andere Entwickler nutzen jedoch die Herangehensweise von AlphaZero und entwickeln ihr eigenes NN nach diesem Ansatz. Der aktuell stärkste Nachfolger heißt Leela Chess Zero (Lc0) [35]. Lc0 hat gemessen an der letzten TCEC-Saison eine Elo von 3586 [31] und ist somit wahrscheinlich stärker als AlphaZero. Lc0 belegte in der 22ten TCEC-Saison den dritten Platz hinter den zwei NNUE-Schachcomputern Stockfish und KomodoDragon [36]. Es bleibt spannend, welcher Ansatz sich durchsetzen wird.

Kapitel 4

NNUE-Implementierung

Ziel dieses Kapitels ist es, Architektur und Implementierung der im Rahmen dieser Arbeit entwickelten NNUE-Evaluationsfunktion zu erläutern. Es wird anfangs auf die Architekturentscheidungen eingegangen. Danach wird geklärt, wie diese Entscheidungen Einfluss auf die Implementierung des Trainers und auf die Integration in einem Schachcomputer haben.

Kapitel Abschnitt 2.1 zeigt, wie die herkömmliche Art und Weise der Positions-Evaluation funktioniert. Verbesserungen der HCE sind nicht einfach. Jeder neue Aspekt in der Evaluationsfunktion muss sorgfältig ausgewählt werden und anschließend per Hand oder mithilfe von Optimierungsalgorithmen, wie z. B. Simultaneous Perturbation Stochastic Approximation (SPSA), angepasst werden [37]. Es ist sehr schwierig, eine Evaluation zu bauen, die für alle möglichen Stellungen optimal ist. Zudem spielt der Bias der Entwickler immer eine Rolle. Die NNUE-Evaluation ist nicht an solche Limitierungen gebunden und kann auf eine ganz andere Art und Weise entscheiden, welche Faktoren wichtig für die Evaluation einer Schachposition sind. Die Entwicklung der Schachcomputerlandschaft zeigt, dass diese Herangehensweise der HCE überlegen ist. Nur in Situationen, in denen es einen klaren Vorteil gibt, ist es sinnvoll HCE zu verwenden. Deshalb verwenden die meisten NNUE-Schachcomputer einen hybriden Ansatz in der Implementierung. In dieser Arbeit wird eine reine NNUE-Evaluation verwendet. Der Grund dafür ist die rudimentäre HCE des verwendeten Schachcomputers. Außerdem tritt dieser Fall bei Schachcomputer gegen Schachcomputer selten auf, da der Vorteil meist klein bleibt.

4.1 Architektur

Die in dieser Arbeit verwendete Architektur ist keine Neue. Sie ist die von Nasu [3] vorgeschlagene Architektur, die ebenfalls in der ersten NNUE-Version von Stockfish verwendet wurde. Sie eignet sich für diesen Prototyp, da sie alle Elemente der NNUE-typischen Architektur enthält und die Grundlage für komplexere Architekturen wie die aktuelle von Stockfish (siehe Kapitel 3) ist. Kleinere Architekturen, wie anfangs in Unterabschnitt 2.4.1 mit 768 Merkmalen, zeigen Verbesserungen über HCE in schwächeren Schachcomputern, wie der in dieser Arbeit verwendete. Jedoch nutzen sie nicht das Potenzial von NNUE aus und tauschen die Spielstärke gegen eine simplere Implementierung.

Die Architektur, welche in dieser Arbeit verwendet wird, ist in Abbildung 4.1 zu sehen. Die Eingabeschicht stellt den dünn besetzten binären Eingabevektor dar. Für das Laden der Daten wird ein Datenlader von Stockfish verwendet, da die verwendeten Trainingsdaten auch von Stockfish sind. Dieser Datenlader hat als Relikt aus der Shogi-Implementierung ein HalfKP-Feature Set mit 41024 statt den für Schach nötigen 40960 Merkmalen [15]. Das hat aufgrund der spaltenweisen Berechnung des Akkumulators keinen Einfluss auf die Performance.

Das Deep Neural Network ist untypisch klein, da die meiste Information in der Feature-Transformator-Schicht enthalten ist [15]. Die übrigen Schichten müssen klein gehalten werden, da sie nicht wie die Feature-Transformator-Schicht mit inkrementellen Aktualisierungen aktiviert werden. Sie werden immer ganz berechnet werden. Die Ausgabeschicht besitzt lediglich ein Neuron, da es sich hier um ein Regressionsproblem handelt. Die Ausgabe des Neurons spiegelt die Evaluation einer Schachposition wider.

Für die Berechnung einer Evaluation ist theoretisch eine Schicht ausreichend. Das limitiert das Netz jedoch sehr stark in der Form der zu modellierenden Funktion. Deshalb werden hier wie üblich für DNNs mehrere Schichten verwendet. Die genaue Zusammensetzung der Schichten ist darauf ausgelegt, dass die Neuronen einer Schicht die Größe der verwendeten SIMD-Register ganz ausschöpfen [3]. Weiter ist es nicht klar, welche Größe sich am besten für den spezifisch verwendeten Schachcomputer eignet. Zur Ermittlung einer optimalen Größe müssen Tests mit verschiedenen Größen durchgeführt werden. Es wird die Größe verwendet, welche bereits Erfolge in anderen Schachcomputern gezeigt hat.

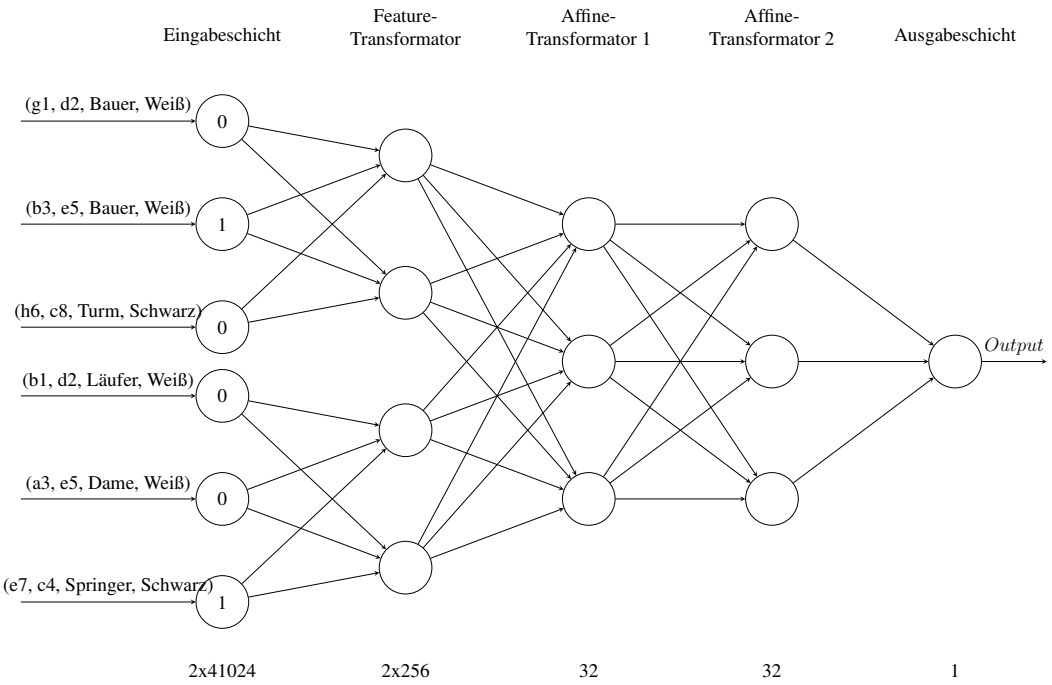


Abbildung 4.1: Das verwendete NN mit einer exemplarischen Eingabe, basierend auf der Abbildung 2.5. Die Bezeichnung der Schichten ist oberhalb und die Anzahl der dazugehörigen Neuronen unterhalb des Netzes zu sehen. Die Eingabeschicht und die Feature-Transformator-Schicht sind in zwei Teile geteilt, basierend auf den zwei Seiten im Schach (Weiß und Schwarz). Sonst sind die Schichten voll vernetzt. Die Ausgabe des Netzes ist eine Evaluation der Position.

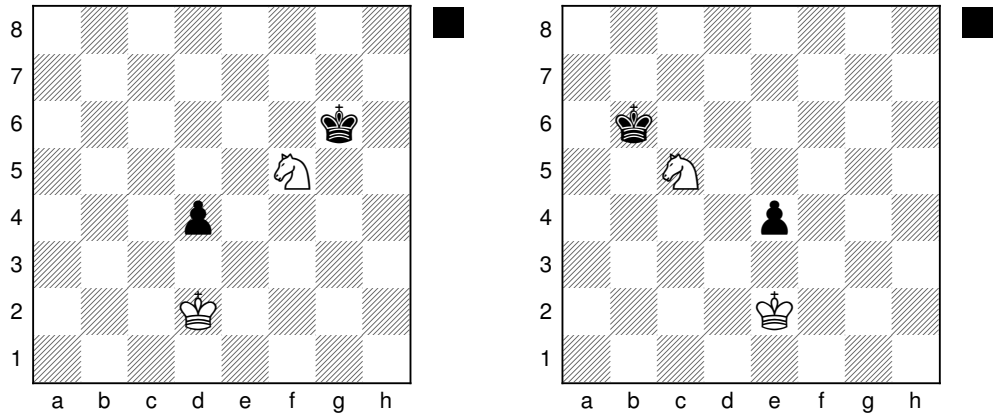
In Abbildung 4.1 ist die Aktivierungsfunktion nicht zu erkennen. Sie ist ein elementarer Bestandteil dafür, dass die Evaluation der Stellung schnell ist. Deshalb wird die ClippedReLU Transferfunktion verwendet. Sie ist nicht linear und kann schnell berechnet werden. Eine Aktivierung mehrerer Neuronen ist mit SIMD gleichzeitig berechenbar. Die Limitierung nach oben ermöglicht die Verwendung von int8 für die Vektoren, die innerhalb des Netzes weitergegeben werden.

4.1.1 Feature-Transformator-Schicht

Die Feature-Transformator-Schicht hat den größten Einfluss auf Geschwindigkeit und Größe [15]. Das gesamte NN hat $10,5$ Millionen trainierbare Parameter. Circa $99,8\%$, also fast alle, sind Teil der Feature-Transformator-Schicht. Das ist untypisch für neuronale Netze. Da aber aufgrund des HalfKP-Feature Sets maximal 30 Merkmale aktiv sein können, ist die tatsächlich verwendete Anzahl an Parametern, pro Zug, deutlich kleiner.

In Abbildung 4.1 ist der Feature-Transformator in zwei Teile geteilt. Einer für Weiß und einer für Schwarz. Je Seite werden 256 Neuronen verwendet. Tatsächlich sind die Gewichte, in dieser Implementierung, für beide Seiten dieselben. Für Weiß werden die Gewichte so wie sie sind verwendet, und für Schwarz wird das Brett um 180 Grad rotiert. Das bedeutet, die schwarzen Figuren werden wie Weiße behandelt und nutzen so die dieselben Gewichte. Eine 180-Grad-Rotation wird wegen des verwendeten Datenladers benutzt.

Eine 180-Grad-Rotation entspricht nicht der Realität, da Schach, im Gegensatz zu Shogi, keine Rotationssymmetrie besitzt. Trotzdem funktioniert es überraschend gut [15]. Alternativ kann das Brett horizontal gespiegelt werden, um die Symmetrie zu wahren. Eine Illustration warum, wie sich Rotation und Spiegelung erhalten ist in Abbildung 4.2 zu sehen. In der Abbildung ist die orientierte Stellung aus Abbildung 2.5 abgebildet. Sie ist dann nötig, wenn schwarz am Zug ist. Die Farben der Figuren werden invertiert. Das ermöglicht es, die Bewertung für Schwarz aus Sicht von Weiß durchzuführen und so dieselben Gewichte zu verwenden. Diese Orientierung findet während der Generierung der Indizierung der Figuren für die Eingabeschicht statt.



(a) Um 180-Grad-Rotiertes Schachbrett **(b)** Horizontal gespiegeltes Schachbrett

Abbildung 4.2: Beispiele für ein orientiertes Schachbrett. Basierend auf Abbildung 2.5. Das Schachbrett ist in zwei Orientierungs-Möglichkeiten gezeigt. Die Schachbretter werden für Schwarz ausgerichtet, das ist der Fall, wenn Schwarz am Zug ist. Die Farben sind invertiert. So können die gleichen Gewichte für weiß und Schwarz verwendet werden.

Außerdem ist es für den Akkumulator wichtig, dass bei der Quantisierung ein Quantisierungsschema verwendet wird, welches einen Überlauf verhindert, egal welche Kombination von Merkmalen aktiv ist [15].

4.1.2 Affine-Transformator-Schichten

Die Affine-Transformator-Schichten sind voll vernetzte Schichten. Sie werden mit der ClippedReLU Transformation aktiviert und dienen dazu dem Netz mehr Tiefe zugeben, ohne dabei zu groß zu werden. Die zwei hier zugehörigen Schichten haben jeweils 32 Neuronen.

Aufgrund der Fehler-Akkumulation der Quantisierung kann es passieren, dass größere/mehr Schichten nahe der Ausgabeschicht einen negativen Effekt haben [15]. Deshalb werden diese Schichten im Vergleich zu der Eingabeschicht sehr klein gehalten.

4.2 Training

Das Training eines NNs beschreibt den Vorgang, bei welchem die Gewichte des Netzes mithilfe von Backpropagation und Gradientenabstieg angepasst werden. Die Gewichte, werden nach dem Training nicht mehr verändert.

Welche Gewichte gewählt werden, hängt von einigen Faktoren ab. In diesem Kapitel wird auf diese eingegangen. Der Trainingsvorgang lässt sich in zwei Teile teilen. Erstens die Auswahl/Generierung der Eingabedaten, welche die Basis des NNs sind. Zweitens der Trainer. Er beschreibt das Programm, welches die Daten einliest und die Gewichte anpasst. Der Trainer ist selbst in weitere Teile untergliedert, die Stellschrauben für das Training darstellen.

4.2.1 Eingabedaten

Die Erzeugung der Eingabedaten ist nicht Teil dieser Arbeit. Jedoch ist es wichtig zu wissen, wie die Eingabedaten generiert werden und wie sie in den Trainer geladen werden, um zu verstehen, wie das NN lernt. Auf die Implementierung des Stockfish Datenladers wird hier ebenfalls nicht eingegangen, da es nicht relevant für das Verständnis eines NNUEs ist.

Im Training für diese Arbeit wurden drei verschieden generierte Datensätze verwendet. Diese Datensätze wurden von Stockfish für das Training der neuesten Variante ihres NNUEs verwendet [38]. Die drei Datensätze unterscheiden sich in der Art und Weise, wie sie generiert wurden. Alle Datensätze sind durch Schachcomputer automatisch, mit Spielen gegen sich selbst, generiert. Einer der Datensätze enthält Spieldaten von Stockfish, aus Spielen mit einer festen Suche von 5000 Knoten pro Zug in Kombination mit dem Unbalanced Human Openings (UHO)-Eröffnungsbuch [39]. Laut Stockfish eignen sich Daten generiert mit Lc0 generell besser für NNUEs [40]. Deshalb besteht einer der drei Datensätze aus eine Mischung von verschieden Lc0 Daten. Trotzdem ist empirisch belegt, dass eine Kombination der Daten zu einem besseren Ergebnis führen. Der dritte Datensatz besteht aus Spieldaten von Stockfish aus der Schachvariante Double Fischer Random Chess (DFRC). DFRC gibt eine zufällige Startposition der Figuren vor. *Double*, weil beide Seiten unterschiedliche Startpositionen der Figuren haben. Der DFRC Datensatz ist für Stockfish interessanter als für diese Arbeit, da Stockfish diesen Modus unterstützt und das mit demselben Netz wie auch herkömmliches Schach. Das ist keine Anforderung für

diese Arbeit, da lediglich traditionelles Schach unterstützt ist. Alle Drei Datensätze enthalten insgesamt 72 Milliarden Positionen.

Es ist nicht klar, welche Daten sich am besten für das Training eines NNUEs eignen [15]. Die für diese Arbeit verwendeten Daten haben sich empirisch als geeignet erwiesen. Sind die gewählten Trainingsdaten zu komplex, ist es schwer zu erkennen, welche die Charakteristiken einer guten/schlechten Position sind. Komplexe Daten sind Evaluationen, die mit einer hohen Suchtiefe generiert werden. Ist die Suchtiefe jedoch zu niedrig, ist die Evaluation ungenau und vermittelt möglicherweise falsche Muster.

Ein weiteres Problem ist es, dass die verwendeten Daten nicht den Anwendungsfall wider Spiegeln. Das Netzwerk soll die aktuelle Stellung so genau wie möglich Evaluieren, die Eingabedaten zeigen jedoch die Evaluationen von Positionen welche z. B. nur 5000 Knoten tief gesucht wurde.

Ein Netz kann mehrmals trainiert werden. Das ist sinnvoll, wenn dabei verschiedene Eingabedaten verwendet werden. Dabei ist die Reihenfolge der Datensätze von Bedeutung. Es kann besser für das Training sein, wenn zuerst simple Daten und dann komplexere Daten verwendet werden. Verschiedene Testergebnisse dazu sind in Kapitel 5 zu sehen. Außerdem kann es sein das ein Trainingslauf, aufgrund der Zufällig initialisierten Gewichte, in einem lokalen Minimum stecken bleibt. Das wird jedoch vernachlässigt, da es deutlich mehr Trainingsläufe bräuchte um diesen Effekt zu verringern.

Eine Epoche ist als 100 Millionen Positionen definiert. Eine Batch, also die Zahl an Positionen, die in einer Iteration verarbeitet werden, beträgt $2^{14} = 16.384$. Die Daten werden nicht sequentiell geladen. Sonst würde eine Batch viele ähnliche Stellungen enthalten, da die Daten aufgrund des verwendeten Datenformats, aufgrund von Datenkomprimierung, ähnliche Stellungen sequentiell speichern. Die Daten werden in zufälligen Schritten übersprungen.

Theoretisch gibt es Daten, die „perfekt“ sind, denn Schach ist für Stellungen mit 7 Figuren gelöst. Das heißt, es gibt eine Datenbank, die das eindeutige Ergebnis (-1, 0, 1) für die Position mit perfektem Spiel kennt. Es ist jedoch nicht sinnvoll, diese Informationen für das Training eines NNUEs zu verwenden. Das Netz ist nicht in der Lage zu verstehen, warum die Stellung gewonnen/verloren ist. Die Konzepte sind oft schwer zu verstehen und der Vorteil wird oft erst in weiter Zukunft realisiert.

Außerdem führt die Verwendung dieser Daten zu einem Bias in der Datenbasis, da sie zu einer Übergewichtung von Endspieldaten führen.

4.2.2 Trainer

Der Trainer übernimmt die Anpassung der Gewichte. Er nimmt die Eingabedaten, aktiviert das Netz vorwärts und passt die Gewichte mit Backpropagation an. Der Trainer muss nicht selbständig implementiert werden, Frameworks, wie in diesem Fall PyTorch, vereinfachen die Implementierung [41]. PyTorch ist ein Python Framework, für das Training von neuronalen Netzen. Es ermöglicht durch vorgefertigte Module die Definition und Training des Netzes auf der GPU.

Zuerst müssen die Schichten des Netzes definiert werden. Die vier Schichten können mit vier Zeilen code definiert werden, wie in Quellcodeverzeichnis 4.1 zu sehen. Dabei handelt es sich um vier dicht vernetzte lineare Schichten.

```
feature_transformer = nn.Linear(41024, 256)
affine_transformer1 = nn.Linear(2 * 256, 32)
```

Quellcodeverzeichnis 4.1: Definition der NNUE-Schichten mit PyTorch, wie auf Abbildung 4.1. Für Schwarz und Weiß werden jeweils dieselben gewichte mittels einer 180-Grad-Rotation verwendet.

Bei der Fehlerrückführung wird zuerst die Vorwärtsaktivierung von der Eingangs- bis zur Ausgabeschicht durchgeführt. Bei diesem NNUE wird die Ausgabe der Feature-Transformator-Schicht per Konkatenation, abhängig von der aktiven Seite kombiniert, da die Ausgabe der Ausgabewert die Evaluation aus Sicht der Aktiven Seite darstellen soll und dasselbe Netz für beide Seiten verwendet werden soll. Die Implementierung dazu ist in Quellcodeverzeichnis 4.2 zu sehen.

```
output = nn.Linear(32, 1)

def forward(us, them, feature_transformer_white, feature_transformer_black):
  feature_transformer_white = feature_transformer(feature_transformer_white)
  feature_transformer_black = feature_transformer(feature_transformer_black)
  # clipped relu activation function (0.0, 1.0)
  feature_transformer_ = torch.clamp(((us * torch.cat([feature_transformer_white,
      feature_transformer_black], dim=1)) + (them * torch.cat([
      feature_transformer_black, feature_transformer_white], dim=1))), 0.0, 1.0)
  affine_transformer1_ = torch.clamp(affine_transformer1(feature_transformer_),
      0.0, 1.0)
```

```
affine_transformer2_ = torch.clamp(affine_transformer2(affine_transformer1_),
    0.0, 1.0)
```

Quellcodeverzeichnis 4.2: Die Vorwärtsaktivierung des in Quellcodeverzeichnis 4.1 definierten Netzes. Die Schichten werden nacheinander, mit der ClippedReLU Transferfunktion, aktiviert. Nach der Feature-Transformator-Schicht werden die beiden Sichtweisen durch Konkatenation kombiniert. Die Eingabedaten werden der Funktion als Parameter gegeben. Welche Seite am Zug ist, wird durch *us* und *them* symbolisiert. Die dünnbesetzten Vektoren *white* und *black* enthalten die Eingabedaten

Nachdem ein Ausgabewert mit dem Netz generiert wurde muss die Genauigkeit dieser Vorhersage bestimmt werden. Dafür wird die Kreuzentropie-Verlustfunktion verwendet. Wird der Fehler auf der Centipawn Ausgabe bestimmt, ist der Gradient sehr groß [15]. Deshalb wird die Ausgabe mit der Sigmoidfunktion zu einer Gewinnwahrscheinlichkeit (0-1) transformiert. Die hier verwendete Kreuzentropie-Verlustfunktion ist von Stockfish übernommen [15]. Sie leitet relativ zu der Ausgabe des Netzes ab, sodass der Fehler mindestens 0 ist. Im Training gibt es außerdem die Besonderheit, das anhand der Evaluation der Position oder über das Ergebnis der gespielten Partie in der Stellung gelernt werden kann. Welches Verhältnis der zwei Zielwerte optimal ist, lässt sich nicht einfach bestimmen. Es ist möglich, dass die Inklusion Ergebnis basierten Trainings gar nicht hilft. Empirische Tests dazu sind in Kapitel 5 zu sehen. Die folgende Implementierung enthält code für dafür. Die Variable *lambda_* steuert das Verhältnis. Ein kleiner Wert (epsilon) sorgt dafür, um $log(0)$ zu verhindern. Zwei Skalierungswerte sorgen dafür, dass die Ausgabe des Netzes eine passende Form für die Gewinnwahrscheinlichkeit eines Schachspiels darstellt. Die Werte hängen von der Schachcomputer Implementierung und den verwendeten Eingabedaten ab [15]. Das Folgende Quellcodeverzeichnis 4.3, zeigt die Implementierung eines Training Schritts, der die Eingabedaten bekommt, eine Vorwärtsaktivierung durchführt, den Fehler mit der Kreuzentropie-Verlustfunktion berechnet und den Fehler zurückgibt:

```
def training_step(self, batch):
  us, them, white, black, wdl_outcome, cp_score = batch
  net_to_cp_score = 600
  scaling = 361

  q = (forward(us, them, white, black) * net_to_cp_score / scaling).sigmoid()
  t = wdl_outcome
  p = (cp_score / scaling).sigmoid()

  epsilon = 1e-12
  p_entropy = -(p * (p + epsilon).log() + (1.0 - p) * (1.0 - p + epsilon).log())
  t_entropy = -(t * (t + epsilon).log() + (1.0 - t) * (1.0 - t + epsilon).log())
```

```
p_loss = -(p * q.log() + (1.0 - p) * (-q).log())
t_loss = -(t * q.log() + (1.0 - t) * (-q).log())
result = lambda_ * p_loss  + (1.0 - lambda_) * t_loss
entropy = lambda_ * p_entropy + (1.0 - lambda_) * t_entropy
loss = result.mean() - entropy.mean()
```

Quellcodeverzeichnis 4.3: Die per Parameter übergebenen Eingabedaten werden vorwärts aktiviert. Danach werden die als CP vorhandenen Werte Skaliert. Anschließend wird die Entropie und der Fehler für Evaluation und Ergebnis ermittelt, sodass mit *lambda_* ein Verhältnis der beiden Einflüsse gebildet werden kann. Schlussendlich gibt die Funktion die Abweichung der Vorhersage von der erwarteten Ausgabe zurück

Mit der Genauigkeit der Ausgabe des Netzes werden mithilfe von Backpropagation die Gewichte angepasst. In dem Training für diese Arbeit wird das adaptive Lernraten-Modell Adadelta [17] verwendet. Adadelta übernimmt die Anpassung der Lernrate und ist bereits von PyTorch implementieren, kann somit sehr einfach, wie Quellcodeverzeichnis 4.4 zeigt, in den Trainer eingebunden werden.

```
torch.optim.Adadelta(parameters)
```

Quellcodeverzeichnis 4.4: Deklaration des Optimierungsalgorithmus in PyTorch. Das PyTorch Modul *optim* bietet verschiedene Optimierungsalgorithmen an. Es wird, die Instanziierung der Adadelta Klasse, dargestellt. Die *paramters* ist eine Liste der Parameter der in Quellcodeverzeichnis 4.1 deklarierten Schichten. Die Lernrate beträgt standardmäßig 1 und wird, durch den Algorithmus, dynamisch angepasst

4.3 Integration in einen Schachcomputer

Folgendes Kapitel beschreibt die Implementierung eines NNUEs in einen Schachcomputer mit dem Fokus auf SIMD-Operationen. Die verwendete Implementierung ist eine auf den Schachcomputer angepasste Version der Stockfish-Implementierung bevor Stockfish zu HalfKAv2 und Buckets gewechselt ist [42]. Da es sich um die gleiche Architektur handelt, bietet es sich an, diese Implementierung zu verwenden. Die Quellcodebeispiele in diesem Kapitel sind für ein besseres Verständnis, vereinfacht. Die Funktionalität, bleibt jedoch bestehen.

Die im Training ermittelten Gewichte, werden in eine Binärdatei gespeichert, von dem Schachcomputer gelesen und in Arrays gehalten. Zum Einlesen der Binärdatei, muss die Integer-Größe bekannt sein. Sie ist durch das gewählte Quantisierungsschema definiert und ist mithilfe von C++ *Templates* leicht anpassbar, falls sich das Quantisierungsschema ändern sollte.

4.3.1 Quantisierungsschema

Eine Quantisierung ist nötig, um CPU-Optimierungen zu ermöglichen. Die als Float trainierten Gewichte und Bias werden bei der Konvertierung des Netzes von einem *.ckpt* zu einer proprietären binären *.nnue*-Datei konvertiert. Alternativ kann die Konvertierung beim Einlesen der Gewichte und Bias in den Schachcomputer stattfinden. Das hat den kleinen Nachteil, dass die Netzwerkdatei etwas mehr Speicherkapazität benötigt. Das hier verwendete Quantisierungsschema ist dem von Stockfish nachempfunden [15]. Es ist daran ausgerichtet, die kleinstmöglichen Integer-Typen zu verwenden. Aufgrund der ClippedReLU-Transferfunktion sind die Gewichte bereits sehr klein. Deshalb werden sie mit bestimmten Faktoren multipliziert, um eine hohe Präzision beizubehalten. In Tabelle 4.1 sind die Faktoren und Datentypen für jede Schicht angegeben. Die zwei linearen Schichten unterscheiden sich nicht und werden deshalb in der Auflistung zusammengefasst. Der Ausgabetyp der Schichten ist ebenfalls aufgelistet. Er ist nicht Teil der Quantisierung der Gewichte und Bias, aber wichtig für das Verständnis. Die Werte müssen so gewählt werden, dass keine mögliche Kombination von aktiven Merkmalen den Akkumulator überlaufen lässt [15].

Tabelle 4.1: Skalierfaktor und Datentypen für Gewichte und Bias des NNUEs sowie Ausgabetyp der Schichten. Basierend auf Werten der Stockfish NNUE Implementierung [15]

Schicht	Gewicht Skalierfaktor	Gewichtstyp	Bias Skalierfaktor	Biastyp	Ausgabetyp
Feature-Transformator	127	int16	127	int16	int8
Affine-Transformatoren	64	int8	8128	int32	int8
Ausgabeschicht	~75,59	int8	9600	int32	int32

Für die Aktivierung der Schichten ändert sich der Bereich der Clipped ReLU von 0 bis 1 zu 0 bis 127. Dadurch ist der Ausgabetyp aller Schichten, die innerhalb des Netzes sind int8. Der Gewichtstyp der Feature-Transformator muss int16 sein, weil er sonst bei der Akkumulation der HalfKP Features überlaufen kann. Das Problem stellt sich bei den Affine-Transformatoren nicht, da durch SIMD-Instruktionen bei der Berechnung des Matrixprodukts der Datentyp automatisch auf int32 angepasst wird. Ein Problem, das hier auftreten kann, ist das Überlaufen der int8-Werte. Deshalb werden sie vor der Multiplikation in beide Richtungen durch $127/64$ begrenzt. Der Bias in den Affine-Transformatoren und der Ausgabeschicht ist hoch, da wir sie mit den int32-Werten der Matrixprodukte addieren und maximale Präzision beibehalten. Die Ausgabeschicht besitzt einen noch höheren Bias-Skalierfaktor, weil die Ausgabe der NNUE-Evaluationsfunktion der HCE gleichen soll (für hybride

Evaluation). Das ist auch wünschenswert für reine NNUE-Implementierungen, um eine Vergleichbarkeit der Evaluation verschiedener Schachcomputer/Versionen zu wahren. Der Gewicht-Skalierfaktor der Ausgabeschicht fällt aus der Reihe, da er nicht ganzzahlig ist. Das liegt daran, dass er sich an dem Bias-Skalierfaktor und dem Aktivierungsbereich (127) anpasst, also $9600/127 = 75,59$.

4.3.2 Feature-Transformator

Die Aufgabe des Feature-Transformators ist es den Akkumulator zu aktualisieren. Eine Aktualisierung ist entweder inkrementell oder eine Neuberechnung des Akkumulators. Der Akkumulator ist ein 64-Byte-Aligned zwei dimensionales Array, welches die Werte aufsummierten Gewichte und Bias der Feature-Transformator-Schicht vor ihrer Aktivierung hält, je Seite hält. Quellcodeverzeichnis 4.5 zeigt den Akkumulator.

```
alignas(64) std::int16_t accumulator[2][256];
```

Quellcodeverzeichnis 4.5: Das Akkumulator Array mit zwei Dimensionen. Die erste Dimension bestimmt die Seite und die zweite das Neuron der Feature-Transformator-Schicht

Eine Neuberechnung des Akkumulators ist in Quellcodeverzeichnis 4.6 gezeigt. Die Funktion erhält das Spielfeld, indem Informationen zu z. B. Figuren und Akkumulator gehalten werden. Wie die Indizes berechnet werden, wird nicht weiter erläutert. Natürlich wird dafür, wie im Training dafür eine 180-Grad-Rotation für die Schwarzen Figuren verwendet. Aufgrund der Column-Major Orientierung der Gewichte in dem Feature-Transformator kann eine Spalte der Gewichtsmatrix direkt für die SIMD-Operation verwendet werden. Da in einem *ymm* Register maximal 16, 16-Bit Integer gleichzeitig verarbeitet werden können, muss diese Operation in 16 Iterationen erledigt werden: Anzahl der Neuronen (256) geteilt durch die Anzahl der berechenbaren 16-Bit Integer (16). Die verwendeten SIMD-Operationen sind simpel, der Akkumulator wird in ein *ymm* Register geladen, mit einer Gewichts-Spalte summiert und wieder in den Akkumulator gespeichert.

```
void refreshAccumulator(Board &board)
{
  std::int16_t(&accumulator)[2][256] = board.state->accumulator;
  for (bool perspective : {WHITE, BLACK})
  {
    IndexList activeIndices;
    AppendActiveIndices(board, perspective, activeIndices);
    std::memcpy(accumulator[perspective], biases, 256 * std::int16_t);
    for (const auto index : activeIndices)
```

```
    {
      const std::uint32_t offset = 256 * index;
      auto accumulation = reinterpret_cast<__m256i *>(&accumulator[perspective]);
      auto column = reinterpret_cast<const __m256i *>(&weights[offset]);
      constexpr std::uint32_t numChunks = 256 / 16;
      for (std::uint32_t j = 0; j < numChunks; ++j)
      {
        __m256i acc = _mm256_load_si256(&accumulation[j]);
        __m256i sum = _mm256_add_epi16(acc, column[j]);
        _mm256_store_si256(&accumulation[j], sum);
      }
    }
  }
}
```

Quellcodeverzeichnis 4.6: Eine Funktion zur Neuberechnung des Akkumulators. Sie nimmt ein *board*, welches Informationen zu den Figuren der aktuellen Position und den Quellcodeverzeichnis 4.5 beschriebene Akkumulator enthält, in welchen ebenfalls das Ergebnis gespeichert wird. Die Berechnung findet für beide Seiten statt. Zuerst werden die Indizes der sich im Spiel befindenden Figuren kalkuliert und der Akkumulator mit den Bias werten initialisiert. Anschließend wird für jeden Index die entsprechende Spalte der Gewichtsmatrix zu dem Akkumulator addiert

Interessanter ist hingegen eine Inkrementelle Aktualisierung. Insgesamt unterscheidet sie sich nicht viel zu der Neuberechnung. Anstatt, dass für alle Figuren die entsprechende Spalte der Gewichtsmatrix addiert wird, werden nur Änderungen mit Addition/Subtraktion berechnet. Auffallend ist, dass laden und speichern in *ymm* Register nicht zu sehen sind. Sie können weggelassen werden, die entsprechenden VMOVDQA Operationen werden automatisch durch den Compiler ergänzt.

```
void updateAccumulator(Board &board)
{
  std::int16_t(&accumulator)[2][256] = board.state->accumulator;
  for (bool perspective : {WHITE, BLACK})
  {
    IndexList removed_indices, added_indices;
    AppendChangedIndices(board, perspective, removed_indices, added_indices);
    constexpr std::uint32_t numChunks = 256 / 16;
    auto accumulation = reinterpret_cast<__m256i *>(&accumulator[perspective]);
    // Difference calculation for the deactivated features
    for (const auto index : removed_indices)
    {
      const std::uint32_t offset = 256 * index;
      auto column = reinterpret_cast<const __m256i *>(&weights[offset]);
      for (std::uint32_t j = 0; j < numChunks; ++j)
        accumulation[j] = _mm256_sub_epi16(accumulation[j], column[j]);
    }
    // Difference calculation for the activated features
    for (const auto index : added_indices)
    {
      auto column = reinterpret_cast<const __m256i *>(&weights[offset]);
      for (std::uint32_t j = 0; j < numChunks; ++j)
```

```
        accumulation[j] = _mm256_add_epi16(accumulation[j], column[j]);
    }
  }
}
```

Quellcodeverzeichnis 4.7: Eine Funktion zur inkrementellen Aktualisierung des Akkumulators. Sie nimmt wie auch die *refreshAccumulator* Funktion aus Quellcodeverzeichnis 4.6 ein *board*. Für beide Seiten, wird abhängig davon, welche Figuren Positionen sich geändert haben, der Akkumulator aktualisiert. Die entsprechenden Spalten der Gewichtsmatrix werden entweder addiert oder subtrahiert, je nachdem ob sich das binäre Merkmal von 0 auf 1 oder von 1 auf 0 geändert hat

In beiden Fällen wird der Akkumulator anschließend mit der ClippedReLU Transformation aktiviert und wie bereits in Unterabschnitt 2.4.2 und Unterabschnitt 4.2.2 erklärt, kombiniert. Dabei werden die 16-Bit Werten des Akkumulators zu 8-Bit Werten transformiert.

4.3.3 Affine-Transformator

Die übrigen Schichten nutzen eine affine Transformation. Das ist das Matrixprodukt der Gewichtsmatrix multipliziert mit dem Eingabevektor (der Ausgabevektor der vorherigen Schicht). Ohne SIMD Anweisungen, ist diese Funktion sehr simpel, siehe Quellcodeverzeichnis 4.8.

```
void affineTransform(const std::uint8_t *input, std::int32_t *output)
{
  for (std::uint32_t i = 0; i < ouputDimensions; ++i)
  {
    const std::uint32_t offset = i * inputDimensions;
    std::uint32_t sum = 0;
    for (std::uint32_t j = 0; j < inputDimensions; ++j)
    {
      sum += weights[offset + j] * input[j];
    }
    output[i] = sum + biases[i];
  }
}
```

Quellcodeverzeichnis 4.8: Funktion für eine Affine-Transformation. Es wird das Matrixprodukt der Gewichtsmatrix (*weights*) multipliziert mit dem Eingabevektor (*input*) gebildet und in einem Ausgabevektor (*output*) gespeichert

Mit der Verwendung von intrinsischen Funktionen wird der Code etwas komplexer. Quellcodeverzeichnis 4.9 zeigt die optimierte Implementierung. Die Summe wird mit 0 implementiert, da die 32-Bit Bias am Ende dazu addiert werden. Für jedes Neuron der Schicht wird das Matrixprodukt gebildet. Das Produkt der 8-Bit Integer ist bei der Multiplikation sind saturated 16-Bit Integer, welche vor der Addition zu

der Summe zu 32-Bit Integern geändert werden. Danach liegen in dem *ymm* Register acht 32-Bit Integer summen vor. Diese müssen für die Ausgabe zusammengefasst werden. Dafür werden Streaming SIMD Extensions 2 (SSE2) Operationen verwendet, welche 16-Byte *xmm* Register verwendet. Zuerst werden die Oberen vier mit den unteren vier addiert. Die resultierenden vier Integer werden zur Verständlichkeit als A bis D bezeichnet. In Zeile 20 werden die Integer so angeordnet, dass A & B sowie C & D summiert werden, dargestellt als AB und CD. Das Ergebnis enthält diese zwei Summen doppelt (AB BA CD DC), je eins davon wird ignoriert. In Zeile 21 wird die Summe von AB und CD gebildet. Diese wird mit dem Bias addiert und in den Ausgabevektor gespeichert.

```
1  void affineTransform(const std::uint8_t *input, std::int32_t *output)
2  {
3    const auto inputVector = reinterpret_cast<const __m256i *>(input);
4    const __m256i ones = _mm256_set1_epi16(1);
5    for (std::uint32_t i = 0; i < ouputDimensions; ++i)
6    {
7      const std::uint32_t offset = i * inputDimensions;
8      __m256i sum = _mm256_setzero_si256();
9      const auto row = reinterpret_cast<const __m256i *>(&weights[offset]);
10
11     constexpr std::uint32_t numChunks = inputDimensions / 32;
12     for (std::uint32_t j = 0; j < numChunks; ++j)
13     {
14       __m256i product = _mm256_maddubs_epi16(&inputVector[j], &row[j]);
15       product = _mm256_madd_epi16(product, ones);
16       sum = _mm256_add_epi32(sum, product);
17     }
18
19     __m128i sum128 = _mm_add_epi32(_mm256_castsi256_si128(sum),
           _mm256_extracti128_si256(sum, 1));
20     sum128 = _mm_add_epi32(sum128, _mm_shuffle_epi32(sum128, _MM_PERM_BADC));
21     sum128 = _mm_add_epi32(sum128, _mm_shuffle_epi32(sum128, _MM_PERM_CDAB));
22     output[i] = _mm_cvtsi128_si32(sum128) + biases[i];
23   }
24 }
```

Quellcodeverzeichnis 4.9: Funktion für eine Affine-Transformation, mit SIMD Operationen optimiert. Der Zusätzliche code entsteht durch Konvertierung der Datentypen. Es wird von 8-Bit Integern zu 32-Bit Integern gegangen.

Für die Affine-Transformator-Schichten wird anschließend noch die ClippedReLU Transformation angewandt. Sie unterscheidet sich leicht von der ClippedReLU in der Feature-Transformator-Schicht, da hier von 32-Bit Integern auf 8-Bit Integer transformiert wird.

Kapitel 5

Ergebnisse

5.1 Testaufbau

Der Testaufbau soll dafür sorgen, dass die hier erreichten Ergebnisse reproduzierbar sind. Außerdem wird in diesem Kapitel die Auswahl der verwendeten Komponenten erläutert.

Schachcomputer sind von Natur aus deterministisch. Deshalb wird zur Vermeidung des immer gleichen Spielablaufs ein Eröffnungsbuch verwendet. Dafür wird das UHO V3 von Pohl [39] zusammengestellte Eröffnungsbuch verwendet. UHO enthält Eröffnungen aus Spielen starker Schachspieler (2300+ Elo), bei denen durch eine Analyse von KomodoDragon [36] ein Vorteil für Weiß vorliegt. Diese Eröffnungen eignen sich gut für Schachcomputer, da so weniger Remis gespielt werden als bei ausgeglichenen Eröffnungen. Konkret werden Eröffnungen mit sechs Zügen und einem Vorteil von +0.90 bis +0.99 für Weiß verwendet. Jede Eröffnung wird von beiden Computern mit beiden Farben gespielt.

Der Schachcomputer unterstützt den Universal Chess Interface (UCI)-Standard. Das ermöglicht die Einbindung in gängige Schachprogramme/GUIs und vereinfacht das Ausführen von Self-Play Turnieren und Turnieren gegen andere Schachcomputer. Tests werden mithilfe der cutechess-cli [43] Konsolen-Anwendung durchgeführt. Die Elo wird ebenfalls mithilfe der cutechess-cli berechnet. Die Elo wird anschließend von Ordo [44] berechnet und anhand von Simulationen werden Abweichungen berechnet, der in folgenden Graphen mithilfe von Errorbalken angegeben ist. Es werden für jede Elo Berechnung mindestens 5000 Spiele gespielt, falls eine höhere Genauigkeit nötig ist, werden mehr Spiele durchgeführt.

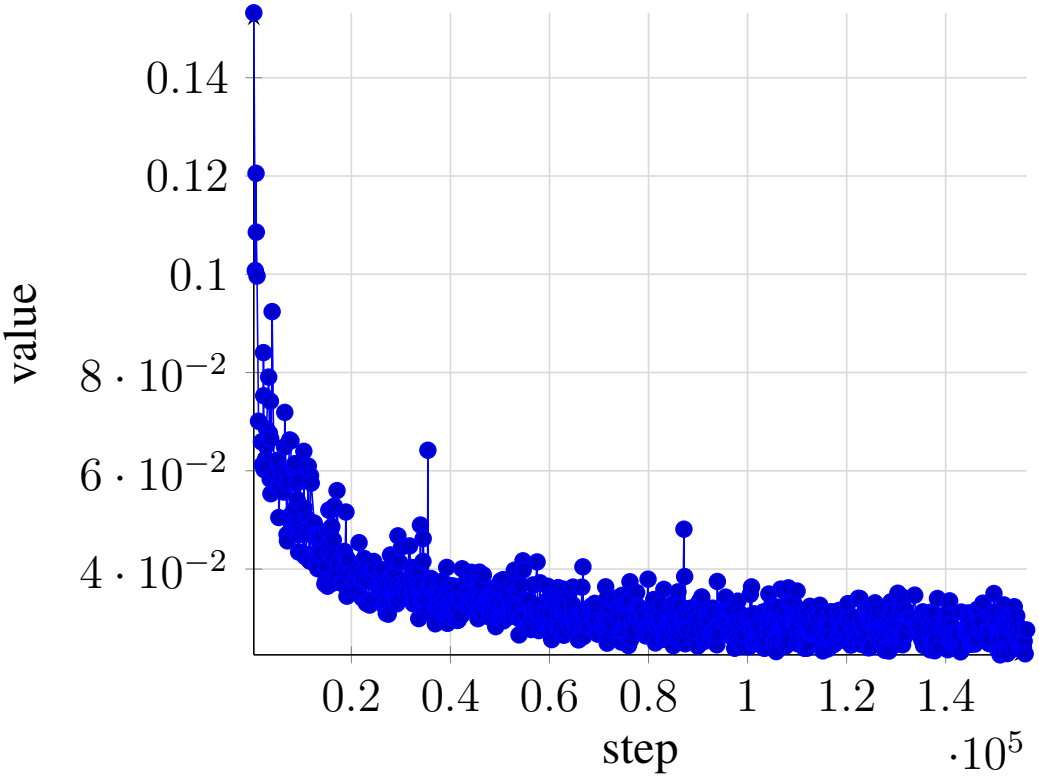


Abbildung 5.1: Verlust über Iterationen.

Die Spiele werden mit Short Time Control (STC) (10s+0,1s) gespielt, außer anders angegeben. Das heißt beide Seiten haben insgesamt 10 Sekunden und erhalten zusätzlich 0,1 Sekunde pro gespieltem Zug.

5.2 Elo-Entwicklung

Unterabschnitt 4.2.2 beschreibt, das die Kreuzentropie-Verlustfunktion verwendet wird. Jedoch wurde in den Grundlagen bereits erläutert, dass es verschiedene Verlustfunktion gibt, welche sich für NNUE eignen. Deshalb wurde ein Test durchgeführt, welche sich besser für den hier verwendeten Schachcomputer eignet.

Unterabschnitt 4.2.1 beschreibt die drei verschiedenen Datensätze die für diese Arbeit verwendet wurden. Unklar ist hingegen in welcher Zusammensetzung und Reihenfolge sie für das Training am besten verwendet werden sollen. Deshalb sind in den folgenden Abbildungen unterschiedliche Zusammensetzungen und Reihenfolgen gezeigt.

Kapitel 6

Diskussion

6.1 Erfolge

6.2 Probleme

Das Validieren des Trainingsfortschritts mithilfe eines Validierungdatensatzes ist nicht möglich, da der Loss auf dem Datensatz basiert und zwischen unterschiedlichen Datensätzen nicht vergleichbar ist. Deshalb ist die Validierung durch das Durchführen von Turnieren der verschiedenen Netze im Vergleich zu dem aktuell stärksten Netz wichtig. Der Nachteil dabei ist, dass die Durchführung der benötigten Spiele für ein verwertbares Ergebnis hoch ist und somit viel Rechenleistung benötigt.

Die Verwendung des UHO-Eröffnungsbuches ergibt normalerweise bei dem Computer-gegen-Computer-Vergleich Sinn, aber da die hier getesteten Schachcomputer aufgrund fehlender Tiefe und Tablebase Schwierigkeiten im Endspiel haben, ist es vermutlich von kleinerer Bedeutung, welche Eröffnungen gewählt wurden.

Die Angaben der berechneten Elo durch Self-Play Turniere ist nicht in tatsächliche Elo gegen andere Computer übertragbar, aber sehr gut für den Vergleich verschiedener Versionen eines Schachcomputers.

Aufgrund

Kapitel 7

Fazit und Ausblick

Abkürzungsverzeichnis

AVX	Advanced Vector Extensions
AVX2	Advanced Vector Extensions 2
AVX512	Advanced Vector Extensions 512
CNN	Convolutional Neural Network
CP	Centipawn
DFRC	Double Fischer Random Chess
DNN	Deep Neural Network
FNN	Feedforward Neural Network
HCE	Hand-Crafted Evaluation
KIS	Künstliche Intelligenz für autonome Systeme
KNN	Künstliches neuronales Netz
Lc0	Leela Chess Zero
MCTS	Monte Carlo Tree Search
MIMD	Multiple Instruction, Multiple Data
MISD	Multiple Instruction, Single Data
NN	Neuronales Netz
NNUE	Efficiently Updatable Neural Network
ReLU	Rectified Linear Unit
SIMD	Single Instruction, Multiple Data
SISD	Single Instruction, Single Data
SPSA	Simultaneous Perturbation Stochastic Approximation
SSE2	Streaming SIMD Extensions 2
STC	Short Time Control
TCEC	Top Chess Engine Championship
UCI	Universal Chess Interface
UHO	Unbalanced Human Openings

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Literatur

[1] Alan Turing, „Faster Than Thought - A Symposium on Digital Computing Machines“. London: Sir Isaac Pitman & Sons, 1953, 1953.

[2] Claude E. Shannon, „XXII. Programming a computer for playing chess“, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, Jg. 41, Nr. 314, S. 256–275, März 1950. DOI: 10.1080/14786445008521796.

[3] Yu Nasu. „NNUE: Efficiently Updatable Neural-Network-based Evaluation Functions for Computer Shogi“. (2018), Adresse: https://www.apply.computer-shogi.org/wcsc28/appeal/the_end_of_genesis_T.N.K.evolution_turbo_type_D/nnue.pdf (besucht am 17. 08. 2022).

[4] Tord Romstad, Marco Costalba, Joona Kiiski und Gary Linscott. „Introducing NNUE Evaluation“. (2020), Adresse: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/ (besucht am 17. 08. 2022).

[5] David Silver, Thomas Hubert, Julian Schrittwieser u. a., „Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm“, 2017. DOI: 10.48550/ARXIV.1712.01815.

[6] Eduardo Vazquez-Fernandez und Carlos A. Coello Coello, „An adaptive evolutionary algorithm based on tactical and positional chess problems to adjust the weights of a chess engine“, in *2013 IEEE Congress on Evolutionary Computation*, IEEE, Juni 2013. DOI: 10.1109/cec.2013.6557727.

[7] Joel Staubach und Marvin Karhan. „Ein Algorithmenbasierter Schachcomputer“. (2022), Adresse: https://github.com/marvinkarhan/chess-engine/blob/master/Karhan_Staubach_Ein_algorithmenbasierter_Schachcomputer.pdf (besucht am 17. 08. 2022).

[8] James R. Slagle und John E. Dixon, „Experiments With Some Programs That Search Game Trees“, *Journal of the ACM*, Jg. 16, Nr. 2, S. 189–207, Apr. 1969. DOI: 10.1145/321510.321511.

[9] David Levy, Hrsg., „Computer Chess Compendium“. Springer New York, 1988. DOI: 10.1007/978-1-4757-1968-0.

[10] Maciej Krawczak, „Multilayer Neural Networks“. Springer, 2013.

[11] Frank Rosenblatt, „The perceptron: a probabilistic model for information storage and organization in the brain.“ *Psychological review*, Jg. 65, Nr. 6, S. 386, 1958.

[12] Jürgen Schmidhuber, „Deep learning in neural networks: An overview“, *Neural Networks*, Jg. 61, S. 85–117, Jan. 2015. DOI: 10.1016/j.neunet.2014.09.003.

[13] Warren S. McCulloch und Walter Pitts, „A logical calculus of the ideas immanent in nervous activity“, *The Bulletin of Mathematical Biophysics*, Jg. 5, Nr. 4, S. 115–133, Dez. 1943. DOI: 10.1007/bf02478259.

[14] Marvin Minsky und Seymour Papert, „Perceptron: an introduction to computational geometry“, 1969.

[15] Joost VandeVondele Thomasz Sobczyk Hisayori Noda, „NNUE“, https://github.com/glinscott/nnue-pytorch/blob/master/docs/nnue.md, 2022.

[16] Sebastian Ruder, „An overview of gradient descent optimization algorithms“, 2016. DOI: 10.48550/ARXIV.1609.04747.

[17] Matthew D. Zeiler, „ADADELTA: An Adaptive Learning Rate Method“, 2012. DOI: 10.48550/ARXIV.1212.5701.

[18] Barry J. Wythoff, „Backpropagation neural networks“, *Chemometrics and Intelligent Laboratory Systems*, Jg. 18, Nr. 2, S. 115–155, Feb. 1993. DOI: 10.1016/0169-7439(93)80052-j.

[19] Douglas M. Kline und Victor L. Berardi, „Revisiting squared-error and cross-entropy functions for training neural network classifiers“, *Neural Computing and Applications*, Jg. 14, Nr. 4, S. 310–318, Juli 2005. DOI: 10.1007/s00521-005-0467-y.

[20] Philipp Gysel, Mohammad Motamedi und Soheil Ghiasi, „Hardware-oriented Approximation of Convolutional Neural Networks“, 2016. DOI: 10.48550/ARXIV.1604.03168.

[21] Jiali Ma, Zhiqiang Zhu, Leyu Dai und Songhui Guo, „Layer-by-Layer Quantization Method for Neural Network Parameters“, in *Proceedings of the International Conference on Industrial Control Network and System Engineering Research*, Ser. ICNSER2019, Shenyang, China: Association for Computing Machinery, 2019, S. 22–26. DOI: 10.1145/3333581.3333589.

[22] Benoit Jacob, Skirmantas Kligys, Bo Chen u. a., „Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference“, 2017. DOI: 10.48550/ARXIV.1712.05877.

[23] Eunhyeok Park, Dongyoung Kim, Sungjoo Yoo und Peter Vajda, „Precision Highway for Ultra Low-Precision Quantization“, 2018. DOI: 10.48550/ARXIV.1812.09818.

[24] Agner Fog, „Optimizing software in C++“, *URL: http://www. agner. org/optimize/optimizing_cpp. pdf*, 2006.

[25] Michael J. Flynn, „Some Computer Organizations and Their Effectiveness“, *IEEE Transactions on Computers*, Jg. C-21, Nr. 9, S. 948–960, 1972. DOI: 10.1109/TC.1972.5009071.

[26] Gang Ren, Peng Wu und David Padua, „A preliminary study on the vectorization of multimedia applications for multimedia extensions“, in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 2003, S. 420–435.

[27] Intel. „Intel Intrinsics Guide“. (2022), Adresse: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (besucht am 29. 08. 2022).

[28] Diogo R. Ferreira, „The Impact of the Search Depth on Chess Playing Strength“, *ICGA Journal*, Jg. 36, Nr. 2, S. 67–80, Juni 2013. DOI: 10.3233/icg-2013-36202.

[29] Simon S. Du, Xiyu Zhai, Barnabas Poczos und Aarti Singh, „Gradient Descent Provably Optimizes Over-parameterized Neural Networks“, 2018. DOI: 10.48550/ARXIV.1810.02054.

[30] Zeyuan Allen-Zhu, Yuanzhi Li und Yingyu Liang, „Learning and generalization in overparameterized neural networks, going beyond two layers“, *Advances in neural information processing systems*, Jg. 32, 2019.

[31] TCEC. „TCEC Leagues Season 22 Engines“. (2022), Adresse: https://wiki.chessdom.org/TCEC_Leagues_Season_22_Engines (besucht am 17. 08. 2022).

[32] E. Korpela, D. Werthimer, D. Anderson, J. Cobb und M. Leboisky, „SETI@home-massively distributed computing for SETI“, *Computing in Science & Engineering*, Jg. 3, Nr. 1, S. 78–83, 2001. DOI: 10.1109/5992.895191.

[33] Stockfish. „Users | Stockfish Testing“. (2022), Adresse: https://tests.stockfishchess.org/users (besucht am 24. 08. 2022).

[34] Garry Kasparov, „Chess, a *Drosophila* of reasoning“, *Science*, Jg. 362, Nr. 6419, S. 1087–1087, Dez. 2018. DOI: 10.1126/science.aaw2221.

[35] UCT/NN AI Community. „Leela Chess Zero: Open source neural network based chess engine“. (2022), Adresse: https://lczero.org/ (besucht am 17. 08. 2022).

[36] Mark Lefler Don Dailey Larry Kaufman. „Dragon by Komodo Chess“. (2022), Adresse: https://komodochess.com/ (besucht am 24. 08. 2022).

[37] James C Spall u. a., „Multivariate stochastic approximation using a simultaneous perturbation gradient approximation“, *IEEE transactions on automatic control*, Jg. 37, Nr. 3, S. 332–341, 1992.

[38] Joost VandeVondele, „Update default net to nn-3c0054ea9860.nnue“, https://github.com/official-stockfish/Stockfish/pull/4100, 2022.

[39] Stefan Pohl. „Anti Draw Openings - The future of Computerchess“. (2022), Adresse: https://www.sp-cc.de/anti-draw-openings.htm (besucht am 26. 08. 2022).

[40] Joost VandeVondele Thomasz Sobczyk Hisayori Noda, „Training datasets“. Adresse: https://github.com/glinscott/nnue-pytorch/wiki/Training-datasets.

[41] Adam Paszke, Sam Gross, Francisco Massa u. a., „PyTorch: An Imperative Style, High-Performance Deep Learning Library“, in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox und R. Garnett, Hrsg., Curran Associates, Inc., 2019, S. 8024–8035. Adresse: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[42] Tord Romstad, Marco Costalba, Joona Kiiski und Gary Linscott, „Stockfish“, https://github.com/official-stockfish/Stockfish, 2014.

[43] Ilari Pihlajisto, Arto Jonsson und Alfred Weyers, „cutechess“, https://github.com/cutechess/cutechess, 2012.

[44] Miguel A. Ballicora, „Ordo“, https://github.com/michiguel/Ordo, 2015.