Inhaltsverzeichnis

1. Einleitung

"Schach ist das schnellste Spiel der Welt, weil man in jeder Sekunde Tausende von Gedanken ordnen muss." (Albert Einstein)

Obwohl ich seit meinem fünften Lebensjahr Schach spiele, bin ich noch immer wie am ersten Tag von der Komplexität des Spiels mit vergleichbar einfachen Regeln, nach denen schon seit Jahrhunderten weltweit gespielt wird, fasziniert.

Während mein Vater das Schachspiel früher in einer engen verrauchten Kneipe von seinem Großvater erlernte, zeige ich heutzutage meiner Großmutter, wie sie mit mir auf Online-Plattformen wie etwa lichess[1] von jedem Ort der Welt aus mit mir Schach spielen kann.

Diese ungeheure Entwicklungsfähigkeit führte auch dazu, dass ich im Jahr 2020 anfing, mich zum ersten Mal wissenschaftlich mit der Brettsportart auseinanderzusetzen. In meinem Projekt "Effizienzanalyse des Minimax-Algorithmus im Bezug auf Schach"[2] beschäftigte ich mich mit der algorithmischen Analyse des Schachspiels. Der technischen Entwicklung und meinem besseren Verständnis der Informatik folgend untersuche ich in diesem Projekt nun die Anwendbarkeit von künstlicher Intelligenz und neuronaler Netzwerke als dessen wichtigste Form[30] auf Schach.

Dabei gehe ich im ersten Teil zunächst auf die theoretischen Grundlagen neuronaler Netze ein, indem ich drei Arten vergleiche und meine Wahl für das Schachspiel detaillierter erläutere. Im Anschluss stelle ich meine Entwicklung einer Webanwendung vor, in der man selbst gegen eine von mir entwickelten Künstlichen Intelligenz Schach spielen und genauer die neuronale Struktur hinter dem Netzwerk verstehen lernen kann. Hier habe ich mich auf den Quellcode und dessen Erläuterung fokussiert.

2. Arten von neuronalen Netzwerken

2.1. Mehrschichtiges Perzeptron

Das mehrschichtige Perzeptron (auch Multilayer-Perzeptron oder engl. multi layer perceptron) als grundlegende Art einer neuronalen Netzwerkstruktur besteht aus einer Eingabeschicht (input layer), einer Ausgabeschicht (output layer) und einer bestimmten Anzahl von versteckten Schichten, die sequentiell aufgerufen werden. Jede Schicht besteht aus zahlreichen Neuronen, die je eine Dezimalzahl speichern können, sowie einer Matrix von Gewichten (weights) und einem Vektor von Neigungen (biases). Auf die genauere Funktionsweise dieses Netzwerks wird in Teil 3 eingegangen; wichtig ist zunächst, dass es sich um das Grundmodell des neuronalen Lernens handelt. Es wurde schon in den 1990er-Jahren benutzt[11] und findet heute noch Verwendung.

Als Basis der NNs kann das Multilayer-Perzeptron prinzipiell in allen Anwendungen genutzt werden, jedoch sind andere Modelle für bestimmte Fälle effizienter oder einfacher in der Nutzung. Für Schach eignet sich dieses Modell, weil eine Schachstellung unproblematisch ins Input-Format hiervon umgewandelt werden kann (siehe 4.2.1) und es keine Bedingungen wie etwa Translationsinvarianz (siehe 2.2) für die Nutzung gibt.

2.2. Konvolutionales neuronales Netzwerk

Eine weitere Netzwerkart ist das konvolutionale neuronale Netzwerk (engl. convolutional neural network, kurz CNN), welches auf dem auf dem mathematischen Prinzip der Konvolution basiert. Dieser wird primär im Bereich der digitalen Bildverarbeitung verwendet. In diesem Kontext bedeutet es, eine Matrix von Zahlen (den sogenannten Kernel) über jedes Pixel des Bilds zu verschieben – die gewichtete Summe des Kernels mit der Nachbarumgebung des jeweiligen Pixels ist das Pixel des neuen Bildes (siehe Abb. 1). Nur durch eine Veränderung des Kernels können die unterschiedlichsten Effekte bzw. Zielbilder

erreicht werden – etwa eine Weichzeichnung (Bewegungs-, Gauss- oder Boxunschärfe), Verschärfung oder sogar Ecken- und Kantenerkennung sind möglich[3]. Für ein mehrfarbiges Bild wird diese Operation für jede Farbschicht ausgeführt.

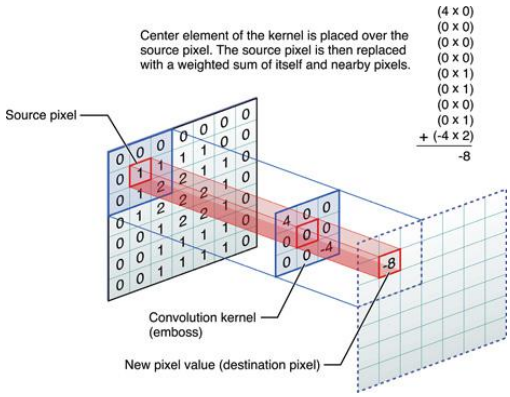

Abb. 1: Eine visuelle Erklärung der Konvolution eines Bilds[4]

Die Parameter einer Schicht des Netzwerks sind die Inhalte dessen Kernels – so muss es nicht für jedes Pixel des Ausgangsbilds mehrere Parameter geben, sondern nur für jedes des Kernels – dies kann eine Reduktion der Parametergröße um ein 10- bis 20-faches hervorrufen[27]. Für Bilderkennung ist dieses Netzwerk ideal, weil diese Art von Aufgaben meist translationsinvariant sind (d.h. wenn ein Gesicht erkannt werden soll, ist die Position im Bild unwichtig). Die Auswertung eines durch mehrfache Konvolution verkleinerten Bildes erfolgt meist durch ein angehängtes Multilayer-Perzeptron[16].

Für die Aufgabe, Schach zu spielen, ist diese Netzwerkart nicht besonders geeignet, da ein Schachbrett mit 8×8 Feldern nicht groß genug ist, um einen Kernel effektiv nutzen zu können. Dazu ist die Translationsinvarianz bei Schach nicht gegeben – ob ein Bauer auf dem Feld g7 oder e3 schlägt, macht einen großen Unterschied.

2.3. Long short-term memory und Rekurrenz

Beim "long short-term memory" (dt. langes Kurzzeitgedächtnis, kurz LSTM) handelt es sich um ein NN, dessen Datenstrom nicht wie bei den bisher vorgestellten Strukturen nur vorwärts verläuft, sondern das auch Daten zwischenspeichern und vergessen kann – diese Fähigkeit wird auch Rekurrenz genannt. So können variabel lange Datenstrukturen, etwa Text oder Audio- bzw. Videoaufnahmen deutlich einfacher eingegeben und verarbeitet werden[28].

Das LSTM setzt sich aus mehreren sogenannten Zellen zusammen, die selbst nicht nur Ein- und Ausgaben, sondern auch einen internen Zustand speichern können, der die zukünftigen Ausgaben (abhängig von den Eingaben) verändern kann. Beispielsweise kann ein solches Netzwerk bei der Einsetzung für Spracherkennung daher nach der Erkennung des Wortes "Hand" eher das Wort "Tasche" als "Flasche" zurückgeben, weil beide diese Worte zwar ähnlich klingen, jedoch "Handtasche" ein gängiger Begriff; "Handflasche" dagegen ein seltenes Kompositum ist. Dies geschieht ohne eine feste Einbindung in die Struktur des Netzwerks – bei einem nicht rekurrenten Netzwerk müsste jede solcher Folgen einzeln gespeichert sein.

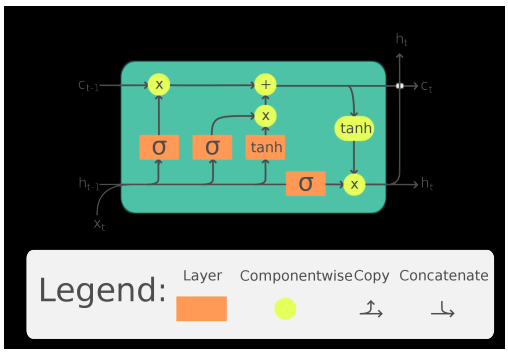

Abb. 2: Eine LSTM-Zelle[5].

LSTM-artige Netzwerke wurden in der Vergangenheit dazu genutzt, um komplexe Videospiele wie etwa das Science-Fiction-Strategiespiel Starcraft oder das Echtzeit-Kampfspiel Dota 2 besser als Menschen zu spielen[6]. Die lange Kette von Eingaben, von denen einige miteinander verknüpft sind, kann mit dieser Art von NN deutlich besser verarbeitet werden als mit Perzeptronen.

Neben LSTM gibt es noch viele andere Arten von rekurrenten neuronalen Netzwerken (RNNs). Diese müssen grundsätzlich nur aus einer Menge von Neuronen bestehen, die beliebig mit anderen Neuronen innerhalb eines Zeitschritts interagieren können – somit können auch zyklische Strukturen gebildet werden. Bei LSTM entstehen diese zyklischen Strukturen nur innerhalb der Zellen, was das Training stark vereinfacht – generell ist das Training eines RNNs sehr komplex. Die Tatsache, dass jeder beliebige gewichtete Graph als RNN betrachtet werden kann, macht den Auswahlprozess im Allgemeinen kompliziert. Aus diesem Grund wird fast immer auf eine spezifische Art eines solchen Modells zurückgegriffen.

Um auf das Schachspiel zurückzukommen: Wenn nur das Brett als Eingabe des Netzwerks erfolgen soll, ist das LSTM sinnlos, da die Eingabe immer die gleiche Länge hat und eine Speicherung von einem Zustand zwischen einzelnen Feldern des Brettes Lernfortschritte eher hindert als unterstützt. Allerdings wäre es auch möglich, stattdessen alle Züge einer Schachpartie als neuronale Eingabe zu nutzen. Weil die Anzahl der Züge variabel ist und Zustandsspeicherung definitiv in diesem Fall notwendig wäre (eine Schachpartie ist schließlich die Akkumulation aller Züge von der Grundstellung aus), kann in diesen Fall ein LSTM genutzt werden. Es ist jedoch unwahrscheinlich, dass dieses zu einer Steigerung der Spielstärke oder Zugqualität führen würde, da der meiste Rechenaufwand darin bestände, die aktuelle Brettposition zu rekonstruieren. Somit ist nicht eindeutig zu beantworten, ob sich eine solche Struktur für das Lernen des Brettspiels eignet und bedürfte noch weiterer Untersuchungen.

3. Das Multilayer-Perzeptron

3.1. Wie ist das Netzwerk aufgebaut?

Die Netzwerkart, die ich hauptsächlich in diesem Projekt nutzen werde, ist das Multilayer-Perzeptron (MLP). Im Gegensatz zu den beiden anderen vorgestellten Modellen ist es universell einsetzbar und daher auch für das Problem des Schachspiels meine Wahl.

Wie bereits in 2.1. erklärt, besteht das Netzwerk aus einer Eingabe- sowie Ausgabeschicht und mehreren versteckten Schichten. Jede dieser Schichten setzt sich aus Neuronen zusammen, die selbst eine Dezimalzahl zwischen 0,0 und 1,0 speichern können (dies wird ihre Aktivierung genannt). Zusätzlich gibt es zwischen jedem Neuron und jedem Neuron der vorherigen Schicht eine Verbindung mit einer Gewichtung, dem sogenannten Weight. Die Aktivierung der Neuronen in einer Schicht wird also deterministisch basierend auf den neuronalen Aktivierungen der vorherigen Schicht berechnet.

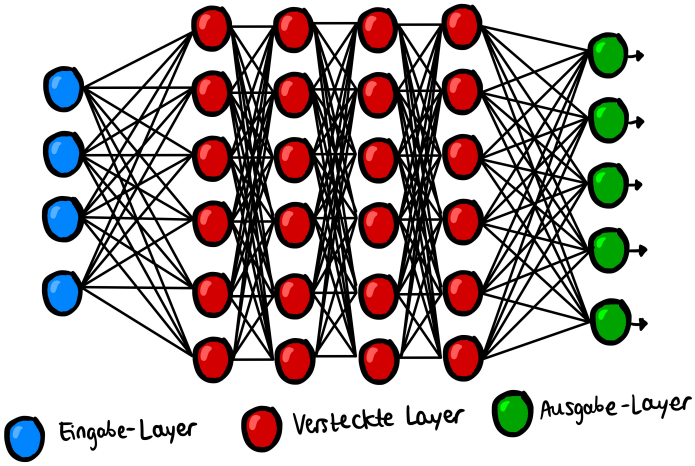

Abb. 3: Eigene Darstellung des MLP

Jede Linie zwischen zwei Neuronen signalisiert hierbei ein Gewicht des Netzwerks.

Um das ganze mathematisch darzustellen, kann jede Schicht als Tripel der aktuellen Aktivierungen der Neuronen, der Weights und der Biases ($a^{(n)}$, W, b) betrachtet werden[8]. Die Aktivierung der Neuronen der nächsten Schicht kann nun als $a^{(n+1)} = f(W * a^{(n)} + b)$ beschrieben werden, wobei f eine Funktion ist, die die Ausgabe der gewichteten Summe (als Ergebnis des Matrix-Vektor-Produkts $W * a^{(n)}$) in den Bereich 0 bis 1 festlegt. Eine erste Auswahl wäre hier die Sigmoid-Funktion $\sigma(x) = (1 + e^{-x})^{-1}$, die stark negativen Eingaben Werte nahe 0 und stark positiven Eingaben Werte nahe 1 ausgibt. Für die meisten Lernprozesse im Bezug auf Schnelligkeit der Kostenminimierung sinnvoller ist allerdings die ReLU-Funktion (kurz für "rectified linear unit", dt. gleichgerichtete lineare Einheit), die kurz als $ReLU(a) = max(a, 0)$ definiert werden kann. Für Eingaben unter 0 gibt die Funktion 0 zurück, ansonsten die Eingabe selbst.

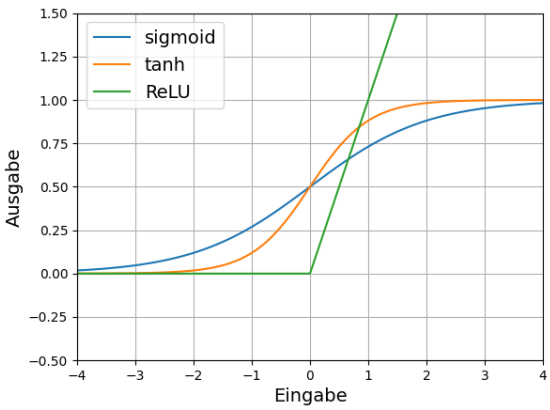

Abb. 4: Unterschiedliche Aktivierungsfunktionen im Vergleich. Bild erstellt mit Python[14] und matplotlib[15]

Unerlässlich ist neben der mathematischen Darstellung auch der Begriff des Parameters. Alle Weights und Biases des Netzwerkes werden grundsätzlich dessen Parameter genannt. Je mehr Parameter ein Netzwerk hat, desto komplexeres Verhalten kann es aufzeigen, aber umso mehr Rechenaufwand entsteht auch bei der Evaluation. Der am häufigsten genannte erste Verwendungszweck eines neuronalen Netzwerks (ähnlich wie bei Programmiersprachen "Hello World") ist das Erkennen von handgeschriebenen Ziffern. Dies ist mit einer Korrektklassifizierungsquote (Anzahl der korrekt klassifizierten Einträge / Anzahl aller Einträge) von über 95% mit einem MLP mit etwa 10000 Parametern lösbar, mit

einem CNN kann man hier die Anzahl der Parameter noch einmal reduzieren. Sehr viel kompliziertere Modelle, wie etwa GPT-3, ein Transformer-Modell, das Text analysieren und schreiben kann, haben über 175 Milliarden Parameter [26]. Jeder Parameter stellt eine Dezimalzahl dar, die verstellbar ist und das Verhalten des Netzwerks ändert.

3.2. Wie kann das Netzwerk lernen?

Bisher wurde beschrieben, wie man ein bereits trainiertes MLP-Netzwerk nutzen kann, um aus Eingabedaten Ausgabedaten zu berechnen. Im nächsten Schritt stellt sich die Frage: wie erhält man ein solches Netzwerk? Anders gefragt: wie werden aus den anfänglich willkürlich gewählten Parameterwerten sinnvolle, die ein Verhältnis zwischen Ein- und Ausgabe repräsentieren?

Das gesamte Netzwerk kann als eine reine Funktion (pure function) betrachtet werden. Eine reine Funktion erfüllt die Bedingung, dass kein externer Zustand verwendet oder modifiert wird, sodass die Ausgabe der Funktion einzig und allein von ihren Eingaben abhängt. Im Falle des Netzwerks hat sie eine bestimmte Anzahl von Ein- und Ausgabeneuronen und ist durch Tausende bis hin zu Millionen von Weights und Biases parametrisiert. Um die optimalen Werte für diese Parameter zu finden, braucht man zunächst Trainingsdaten. Tatsächlich ist häufig das Fehlen von Trainingsdaten der Hauptgrund, warum Startups und Projekte, die maschinelles Lernen mithilfe von neuronalen Netzwerken einsetzen, fehlschlagen[9] [10].

Sobald allerdings genug Daten – in ausreichender Quantität und Qualität – vorliegen, kann man anfangen, eine Verlustfunktion aufzustellen. Die Verlustfunktion ist eine Funktion, die als Argumente das komplette Netzwerk (d.h. die Liste aller Parameter) sowie eine neuronale Eingabe und die gewünschte oder im Idealfall erzielte Ausgabe des Netzwerks erhält. Eine einfache solche Funktion ist "mean squared error" (kurz MSE), die für alle Ausgabeneuronen das Quadrat der Differenz zwischen der tatsächlichen und erwünschten Ausgabe berechnet und summiert[12]. Man definiert nun die Kostenfunktion (oder Zielfunktion) als Durchschnitt der Verlustfunktion über alle Trainingsdaten. Bei einem komplett untrainierten Netzwerk, dessen Parameter noch willkürliche Werte haben, wird dieser Wert sehr hoch liegen; ein trainiertes Netzwerk hat dagegen einen geringen Wert.

Das Ziel des Trainings – das, was üblicherweise als maschinelles Lernen bezeichnet wird – besteht nun darin, ein Minimum dieser Kostenfunktion zu finden. Das so gefundene Modell wäre das, welches die Eingabedaten des Trainings am besten den richtigen Ausgabedaten zuordnet, sie also korrekt klassifiziert. Bei einer simplen ein- oder zweidimensionalen Funktion könnten einfach die Nullstellen der Ableitung berechnet werden, doch dies ist bei Tausenden bis hin zu Milliarden von Parametern einfach nicht mehr tragbar.

Stattdessen muss ein Verfahren angewandt werden, das sich Gradientenabstieg nennt. Dabei wird die n-dimensionale Ableitung an der aktuellen Stelle (den aktuellen Parametern des Modells) berechnet – es entsteht ein n-dimensionaler Vektor, bei dem jedes Element einen Parameter referenziert. Dieser sagt für jeden Parameter aus, wie stark eine Veränderung dessen für eine Veränderung der Kostenfunktion sorgen würde. Wenn man den Vektor mit -1 multipliziert und mit einer Konstante verkleinert (um nicht zu viele Veränderungen am Modell auf einmal durchzuführen), erreicht man, dass die Kostenfunktion verkleinert wurde. Wendet man diesen Schritt viele Male an, wird ein lokales Minimum der Funktion erreicht – das Modell kann nun also für mehr Trainingseingaben richtigen Ausgaben vorhersagen[11].

Weil es sehr rechenaufwändig ist, die Kostenfunktion über alle verfügbaren Trainingsdaten zu definieren, werden diese meist in Stapel (batches) aufgeteilt, die dann separat durchgegangen werden. Dies verhindert zum einen, dass der Gradientenabstieg bei einem kleinen lokalen Minimum stoppt und nicht mehr weiter sucht, da die Funktion stets eine andere ist, und ermöglicht zum anderen in der Praxis einen deutlich schnelleren Trainingsablauf[8].

Um die tatsächliche Lernrate des Modells zu überprüfen, muss neben den Trainingsdaten noch ein weiterer Datensatz, die sogenannten Validierungs- oder Testdaten, bereitgelegt werden. Dieser wird nach dem Training dazu genutzt, um die tatsächliche Genauigkeit (accuracy) zu testen. Es ist wichtig, die Trainingsdaten klar von den Validierungsdaten zu trennen – sonst könnte man ein Modell, das sich nur die Muster in den Trainingsdaten merkt (sog. überangepasstes Modell), nicht von einem anderen Modell, das wirklich die Muster der Eingabe erkennen kann und verstanden hat, unterscheiden[8] [11].

Häufig wird dazu ein Lernprozess in mehrere Epochen aufgeteilt, nach denen die Korrektklassifizierungsrate des Modells an den Trainings- und Validierungsdaten getestet und miteinander verglichen wird. So kann man nachher kontrollieren, ob eine Überanpassung, d.h. eine Abweichung der Genauigkeit von Trainings- und Testdaten, stattgefunden hat und das Netzwerk entsprechend anpassen. Es gibt verschiedene Möglichkeiten, eine solche Überanpassung zu verhindern – die optimale ist, mehr Trainingsdaten zu beschaffen, aber wenn dies keine Option ist, kann zur Kostenfunktion ein Bestrafungsfaktor für zu hohe Gewichtungen addiert werden, oder ein Dropout-Faktor kann genutzt werden, der Neuronen einer Schicht zufällig ausschaltet und so verhindert, dass Daten nur gespeichert und nicht analysiert werden[11] [12].

Um das Training kürzer beschreiben zu können, wird der Begriff des Hyperparameters genutzt – so nennt man Einstellungen, die das neuronale Netzwerk auf einer höheren Ebene als die trainierbaren Parameter betreffen. Sichtbare Hyperparameter sind etwa die Anzahl der Ein- und Ausgabeneuronen, unsichtbare (nach Abschluss des Trainings) hingegen die Anzahl der versteckten Schichten und Neuronen in diesen Schichten. Auch die Größe der Stapel oder die Anzahl der Epochen können als unsichtbare Hyperparameter betrachtet werden, auch wenn sie nicht das tatsächliche Modell, sondern nur das Training betreffen.

4. Die eigene Entwicklung des Programmcodes

4.1. Welche Trainingsdaten werden verwendet?

Wie bereits in Punkt 3.2 beschrieben, ist der wichtigste Faktor für das neuronale Lernen die Verfügbarkeit von Daten. Für das Schachspiel gibt es glücklicherweise die Lichess-Datenbank[17], die über vier Milliarden Schachpartien für die freie Nutzung unter CC0[18] zur Verfügung stellt. Wie bereits in der Einleitung erwähnt ist lichess [1] eine Schachwebsite, die den Prinzipien der freien Software folgt (und folglich völlig quelloffen ist) und auf der in jeder Minute mehr als 2000 Schachpartien begonnen werden. Zusammen mit der großen Datenmenge sind also zwei der "drei Vs" von Big Data (volume, velocity und variety)[19] erfüllt. All diese Schachpartien werden in der genannten Datenbank gespeichert und können zu jedem möglichen Verwendungszweck genutzt werden.

Für das maschinelle Lernen ist das natürlich ideal, weil mehr als hinreichend viele Trainingsdaten vorliegen und somit Über- oder Unteranpassung (nicht ausreichend viele Trainingsdaten für ein gutes Ergebnis) keine Probleme mehr darstellen. Die bereitgestellten Dateien liegen im PGN-Format (kurz für Portable Game Notation), einem menschenlesbaren Plain-Text-Format, vor und müssen somit erst in neuronale Ein- und Ausgaben konvertiert werden.

4.2. Welche Form sollte das neuronale Netzwerk haben?

Nachdem ich den Netzwerktyp ausgewählt habe, muss festgelegt werden, wie das Netzwerk aufgebaut ist.

4.2.1. Eingabe

Als Eingabe eignet sich für ein MLP die Schachstellung (im Gegensatz zu einem LSTM-Netzwerk, bei dem die gesamte Partie und ihre Züge eingegeben werden könnte). Also muss diese nun in eine Liste von Neuronen umgewandelt werden. Eine Möglichkeit wäre,

jeder Figur eine Zahl zwischen 0,0 und 1,0 zuzuweisen und somit ein Neuron pro einem der 64 Felder zu benutzen. Dies ist allerdings nicht empfehlenswert, da es für das Netzwerk aufwändig wäre, die Werte wieder voneinander zu trennen. Stattdessen habe ich für jede mögliche Aufstellung eines Feldes ein Neuron gewählt – pro Feld entstehen also folgende dreizehn Neuronen: *leer*, Bauer (weiß), Bauer (schwarz), Springer (weiß), Springer (schwarz), Läufer (weiß), Läufer (schwarz), Dame (weiß), Dame (schwarz), König (weiß) und König (schwarz). Um die Konvertierung zwischen Schachstellung und Eingabe zu vereinfachen, lege ich das leere Feld an den Anfang und fahre dann mit den weißen und schwarzen Steinen fort.

Die wichtigste Information neben der Position von Figuren auf dem Brett ist, welcher Spieler den nächsten Zug ausführt. Diese zusätzliche Information wird nach der Information über alle Felder als ein Neuron, dessen Aktivierung Weiß und Deaktivierung Schwarz bedeutet, angehängt. Insgesamt habe ich nun also 64 * (2 * 6 + 1) + 1 = 833 Eingabeneuronen, wobei 64 die Anzahl der Felder und 6 die Anzahl der verschiedenen Spielsteine pro Spieler (Bauer, Springer, Läufer, Turm, Dame und König) ist. Alle diese Neuronen können entweder deaktiviert (0,0) oder aktiviert (1,0) sein, was den jeweiligen Zustand repräsentiert.

4.2.2. Ausgabe

Die Frage nach dem Format von Ausgabeneuronen ist nicht so einfach zu beantworten, weil es mehrere Möglichkeiten gibt – drei hiervon habe ich bisher näher untersucht. Häufig haben MLPs als Ausgabe das gleiche Format wie für die Eingabe, was die Verarbeitung der Ergebnisse vereinfacht – in diesem Fall würde dies bedeuten, erneut 833 Neuronen auszugeben, die das neue Schachbrett nach dem gespielten Zug repräsentieren. Dieser Entwurf hat jedoch viele Nachteile. Zunächst einmal müsste jede versteckte Schicht die komplette Stellung immer speichern, damit kein Zustand verloren gehen kann – der Fokus würde also weniger darauf liegen, zu lernen, einen guten Zug auszuführen und mehr darauf, die Eingabedaten möglichst komprimiert zu speichern und wiederzugeben. Außerdem hat ein solches Netzwerk einfach zu viele Möglichkeiten und es ist nicht klar, was genau einen gespielten Zug ausmacht und was nur einfach zur falschen Repräsentation der Stellung gehört. Deshalb kann diese Methode verworfen werden.

Stattdessen sollte sich die Ausgabe nur auf den Schachzug selbst fokussieren. Menschen schreiben Züge meist in der Standard Algebraic Notation, kurz SAN, auf. Diese beginnt mit einem Kürzel für die Figur (etwa S für Springer oder D für Dame) und endet mit dem Feld, auf welches diese Figur gestellt wird. Danach wird noch ein Symbol eingefügt, falls es sich um ein Schach oder Schachmatt handelt – der komplette SAN-Zug könnte etwa `Df7#` oder `e4` (Bauern haben kein Kürzel) lauten. So kann der Gedankengang, mit dem Menschen Züge im Schach ziehen (etwa: Dame nach f7, Schachmatt), präzise in ein Textformat konvertiert werden.

Dies ist für Maschinen aber, wenn die Züge nicht gerade einem Nutzer oder einer Nutzerin präsentiert werden, eher irrelevant und UCI (Universal Chess Interface) wird sehr viel häufiger benutzt. In dieser Art und Weise, Züge zu notieren, wird einfach nur das Anfangsfeld und das Endfeld nacheinander aufgeschrieben (etwa `h5f7` oder `e2e4`). So kann jeder Zug notiert werden (mit der Ausnahme von Umwandlungen, aber da über 97%[20] aller Umwandlungen zur Dame geschehen, kann dies erst einmal vernachlässigt werden).

Dieser Notation folgend habe ich zuerst ein Datenformat mit 128 Ausgabeneuronen benutzt (die ersten 64 verschlüsseln das Feld, von dem aus der Zug startet und die letzten das, an dem er endet). Dies hat sich später als ineffizient herausgestellt, da, um den tatsächlich spielbaren Zug zu finden, die vordere und hintere Hälfte miteinander kombiniert werden müssten. Auch das Training erfolgt schwieriger als nötig, weil das Netzwerk immer zwei maximale Aktivierungen haben sollte und somit als Kostenfunktion entweder MSE (siehe 3.2) oder eine binäre Kreuzentropie genutzt werden müsste – dies ist für das Finden von Schachzügen nicht notwendig, weil diese Algorithmen darauf ausgelegt sind, Mehrfachklassifizierungsprobleme mit mehreren Ausgaben zu optimieren. So kann nicht darauf geachtet werden, dass immer exakt eine Ausgabe (der Schachzug) erfolgen soll und wertvolle Rechenzeit wird vergeudet.

Sinnvoller ist die Kodierung als einfaches Klassifizierungsproblem mit 4096 Labels. Man kann alle möglichen Züge auflisten und lexikographisch sortieren, sodass man ["a1a1", "a1a2", "a1a3", "a1a4", ..., "a1a8", "a1b1", ... "a1h7", "a1h8", "a2a1", "a2a2", "a2a3", ... "a8h8", "b1a1", "b1a2", ..., "h8h6", "h8h7", "h8h8"] als Liste aller Kombinationen zweier Felder erhält. Es ist klar zu erkennen, dass viele davon keine legalen Züge sein können – etwa kann kein Zug auf dem gleichen Feld starten und enden – jedoch ist dies erst einmal nicht relevant, da das Ziel des neuronalen Lernens auch ist, nicht auftretende Züge nicht auszugeben. Anders ausgedrückt: Es ist nicht möglich, nur legale Züge als Ausgaben anzubieten (von denen es in einer gegebenen Situation meist 5-30 gibt), da das Netzwerk universell für alle Stellungen nutzbar sein soll. Also verwende ich (zunächst) alle 4096 (Zug-)Möglichkeiten.

Durch den Fakt, dass ein hypothetisches optimales Netzwerk, das allen Eingaben stets ihre richtigen Ausgaben zuordnet, immer genau einen Zug ausgeben soll, kann man nun als Verlustfunktion die kategoriale Kreuzentropie benutzen, die verlässlichere Ergebnisse erzielt, weil sie genau auf diesen Fall optimiert ist.

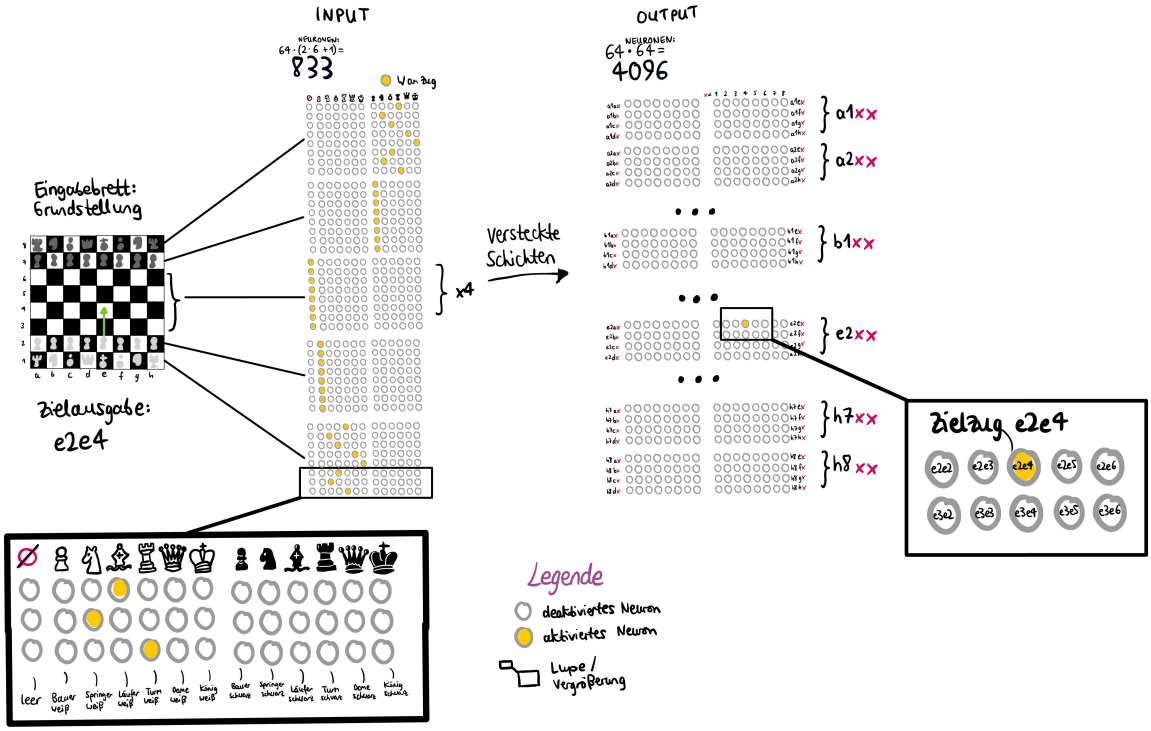

Abb. 5: Eigene Darstellung meines finalen Modells

4.3. Wie werden die Daten vorbereitet?

Nun weiß ich also, wie die sichtbaren Hyperparameter des Modells aussehen und muss Trainingsdaten erstellen, die das gleiche Format haben. Dazu kann ich erst einmal eine von den über hundert Dateien aus der Lichess-Datenbank herunterladen und extrahieren – man erhält eine einige Gigabyte große PGN-Datei, die Schachpartien im menschenlesbaren Format speichert. Weil das Ziel ist, Millionen von Daten zu verarbeiten, ist die Ausführgeschwindigkeit des Programmes von höchster Bedeutung – deshalb habe ich mich hier für die Programmiersprache Rust[29] entschieden. Rust ist speichersicher, von der Schnelligkeit auf einem Niveau mit C oder C++ und ermöglicht es, auf die Daten funktional und bedarfsorientiert (in der Fachsprache als 'lazy evaluation' bezeichnet) zuzugreifen. Dies ist besonders wichtig, da es sonst unmöglich werden könnte, alle Daten gleichzeitig in den Arbeitsspeicher zu legen – ein erster Prototyp, der von mir in Python geschrieben wurde, hatte genau dieses Problem und die Verarbeitung von mehr als einer Million Brettern wurde unmöglich.

Die wichtigsten hier verwendeten Bibliotheken (in Rust auch "crates" genannt) sind:

- pgn_reader, ein schlanker PGN-Parser, der mit dem Besuchermuster (visitor pattern) arbeitet und so zeitsparend und effizient arbeiten kann. pgn_reader liest immer so viel aus einer Datei, wie gerade benötigt wird – so spart man unnötige Dateizugriffe und damit Zeit und Ressourcen.
- shakmaty, eine simple Schachimplementation zur Validierung und Ausführung von Zügen.
- npyz, eine Bibliothek zum Schreiben und Lesen von .npy-Dateien. In diesem Dateiformat werden die vorbereiteten Trainingsdaten gespeichert, um optimal in Python (und insbesondere der Python-Bibliothek numpy) gelesen werden zu können.

```rust
struct NeuralInputCreator {
    board: Chess,
    moves: Vec<(Chess, Move)>,
    considerable_game: bool,
}

impl Visitor for NeuralInputCreator {
    type Result = Option<[(Chess, Move); 10]>;
    // ...
}
```

Ein struct (Produkttyp in Rust) namens NeuralInputCreator, der drei Felder beinhaltet, wird deklariert. `board` speichert die aktuelle Brettposition, `moves` die gespeicherten Züge zusammen mit ihren Positionen und `considerable_game`, ob dieses Spiel überhaupt genutzt werden soll. In meinem Beispiel beinhalten die Trainingsdaten nur Spiele von Spielern, die eine lichess-Wertung von mehr als 1700 haben, was der Spielstärke eines fortgeschrittenen Vereinsspielers entspricht. Alle anderen Partien werden sofort verworfen.

Der Typ `Result` in der Implementation des Visitor-Patterns ist ein optionaler Array mit 10 Elementen, deren Typ ein Tupel bestehend aus einer Schachstellung (`Chess`) und einem Zug für diese Stellung (`Move`) ist.

```rust
impl Visitor for NeuralInputCreator {
    // ...
    fn san(&mut self, san_plus: SanPlus) {
        if !self.considerable_game {
            return;
        }
        match san_plus.san.to_move(&self.board) {
            Ok(m) => {
                self.moves.push((self.board.clone(), m.clone()));
                self.board.play_unchecked(&m);
            }
            Err(e) => {
                eprintln!("Error: {}", e);
            }
        }
    }
    // ...
}
```

Die `san()`-Methode im Trait Visitor wird dann für jeden SAN-Zug des PGN-Dokuments aufgerufen. Falls der Zug gültig ist, wird er zusammen mit der aktuellen Brettposition in die Liste gespeicherter Züge hinzugefügt und anschließend auf dem Brett ausgeführt. Ansonsten wird eine Fehlermeldung ausgegeben.

```rust
impl Visitor for NeuralInputCreator {
    // ...
```

```
    fn end_game(&mut self) -> Self::Result {
        if !self.considerable_game {
            return None;
        }
        let rng = rand::thread_rng();
        self.moves
            .choose_multiple(&mut rng, 10)
            .cloned()
            .collect_vec()
            .try_into()
            .ok()
    }
}
```

Die `end_game()`-Methode wird am Ende eines Spiels aufgerufen. In dieser werden 10 zufällige Bretter zusammen mit ihren gespielten Zügen ausgewählt und in einem Array zurückgegeben. So wird sichergestellt, dass immer 10 Elemente zurückgeben werden. Falls weniger als 10 Züge vorliegen oder das Spiel gar nicht gezählt werden soll, wird `None` (also die Abwesenheit eines Werts) zurückgegeben. Weil Rust eine sichere Sprache ist, gibt es keine null-Pointer oder ähnliche Risiken, stattdessen muss explizit mithilfe des monadischen `Option<T>`-Typ angemerkt werden, dass ein Wert nicht vorhanden sein könnte.

```
fn save_boards(io_pairs: impl Iterator<Item = (Chess, Move)>) -> io::Result<()> {
    // ...
    let io_pairs_chunked = io_pairs
        .map(|(chess, r#move)| (chess_to_input(&chess), move_to_output(&r#move)))
        .unique_by(|(input, _)| {
            let mut hasher = DefaultHasher::new();
            input.hash(&mut hasher);
            hasher.finish()
        })
        .chunks(1_000_000);
    // ...
}
```

Die `save_boards`-Funktion, die von der Hauptfunktion aus aufgerufen wird, nimmt die Input-Output-Paare (von Brettern und Zügen) als Parameter an. In der neuen Bindung, `io_pairs_chunked`, werden den Brettern und Zügen ihre neuronalen Repräsentationen zugewiesen und sie daraufhin basierend auf einem Hashwert auf Eindeutigkeit sortiert. Die `unique_by`-Methode des Iterators nutzt intern ein `HashSet`, das die Hashwerte speichert und bereits gesehene Werte überspringt. Falls die Elemente nicht zuerst gehashed würden, hätten man die gleichen Arbeitsspeicherprobleme wie in Python.

```
fn chess_to_input(chess: &Chess) -> [bool; 833] {
    // Ein (veränderbarer) Array von Wahrheitswerten mit 833 Elementen wird erstellt
    let mut output = [false; 833];

    output[0] = chess.turn().is_white(); // Das erste Neuron verschlüsselt die Zugfarbe

    let board = chess.board().clone();
    for (index, square) in Square::ALL.into_iter().enumerate() { // Für jedes Feld des Bretts:
        // Der Grundindex ist die Zahl von 0 bis 64 mal 13 plus 1.
        let mut output_index = index * (1 + 2 * 6) + 1;
        output_index += match board.piece_at(square) {
            None => 0, // Ein leeres Feld hat den Offset 0,
            Some(Piece { color, role }) => {
                (match color {
                    Color::White => 0usize,
                    Color::Black => 6,
                } + u32::from(role) as usize)
```

```
                // ein besetztes das der Rolle (1 bis 6) plus der Farbe (0 oder 6).
            }
        };
        // An der so berechneten Stelle wird `true` eingesetzt.
        output[output_index] = true;
    }

    output // und geben den Array zurück.
}

fn move_to_output(m: &Move) -> u16 {
    // Das Herkunfts- und Ankunftsfeld des Zugs wird berechnet.
    let (from, to) = match m {
        Move::Normal { from, to, .. } | Move::EnPassant { from, to, .. } => (*from, *to),
        Move::Castle { king, rook } => // Sonderfall der Rochade (hier ausgespart)
        Move::Put { .. } => unreachable!(),// Put-Züge sind auf Lichess und im Schach verboten
    };
    // Mithilfe der `.into()`-Methode kann die Reihe in einen Byte konvertiert werden.
    let from: u8 = from.into();
    // Diese ist generisch über ihren Rückgabewert.
    let to: u8 = to.into();
    // Am Ende berechnen wird 64 * from + to berechnet, um eine eindeutige Zahl zwischen
    // 0 und 4095 zu erhalten, die den Zug repräsentiert.
    // Weil 4095 größer als 255, das Limit für einen `u8`,
    // ist, erstellt man zuerst einen `u16`, damit kein Integer-Overflow geschieht.
    from as u16 * 64 + to as u16
}
```

> Die beiden Methoden `chess_to_input` und `move_to_output` nehmen je eine Stellung bzw. einen Zug (aus der shakmaty-Bibliothek) als Eingabe und berechnen die neuronale Repräsentation. Die Funktionsweise wird hier nicht noch einmal ausführlich erläutert, weil der Quelltext vollständig dokumentiert ist.

Durch das Nutzen dieser Rust-Funktionen kann ich aus der PGN-Datenbank mehrere Trainingsdateien erstellen, die in meinem Fall je eine Million Input- oder Output-Beispiele enthalten. Die Input-Dateien haben also in Numpy die Shape `(1_000_000, 833)` mit dem Datentyp `bool`, die Output-Dateien, die simpel kategorial mithilfe von Zwei-Byte-Ganzzahlen dargestellt werden, die Shape `(1_000_000,)` mit dem Datentyp `u16`.

4.4. Wie wird das neuronale Netzwerk erstellt, trainiert und getestet?

Für das Erstellen des Netzwerks nutze ich die Python-Bibliothek Keras[21]. Keras erlaubt, neuronale Netzwerke in Python syntaktisch präzise sowie elegant zu modellieren (ohne viel Boilerplate wie erforderte Klassen oder schwer zu verstehende Methoden, dt. Kesselblech) und basiert dabei auf der für alle Formen von maschinellem Lernen verbreitete TensorFlow-Bibliothek[22].

Mit Keras kann ich das Modell wie folgt definieren:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_shape=(833,)))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(4096, activation='softmax'))

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
```

Mithilfe von Keras erstelle ich ein sequentielles Modell. Da alle zum Modell hinzugefügten Schichten den Datentyp `Dense` haben, handelt es sich um ein MLP. Der Eingabetyp hat die Shape `(833,)`, was bedeutet, dass es 833 Neuronen gibt, die in einer Dimension vorliegen.

Alle Schichten mit Ausnahme der letzten nutzen ReLU als Aktivierungsfunktion, da dies die Trainingsgeschwindigkeit im Vergleich zum Sigmoid erhöhen kann. Diese nutzt dagegen Softmax, um zum Schluss jedem Neuron eine Wahrscheinlichkeit zwischen 0 und 1 zuzuordnen, die insgesamt zu 1 addieren. Dies erleichtert die spätere Ausgabe und Datenverarbeitung, das Training wird nicht merklich beeinträchtigt, da die Funktion recht einfach zu berechnen ist. Diese Layer hat 4096 Neuronen, die den finalen Zug (wie in 4.2.2 erklärt) enkodieren.

Zum Schluss kompiliere ich das Modell mithilfe des Adam-Optimierers (der das Gradientenabstiegsverfahren (siehe 3.2) implementiert), als Verlustfunktion wähle ich die kategoriale Kreuzentropie. Außerdem lege ich fest, dass die Genauigkeit des Netzwerks (accuracy) beim Training und Test neben der Verlustfunktion als Messwert aufgezeichnet und dargestellt wird.

```
history = model.fit(
    np.load("train_input.npy"),
    np.load("train_output.npy"),
    epochs=30,
    validation_data=(np.load("validation_input.npy"), np.load("validation_output.npy")),
)
```

Die `.fit()`-Methode führt das Training aus. Nach jeder Epoche wird das trainierte Modell anhand der Validierungsdaten auf eine Überanpassung überprüft. Die Verlust- und Genauigkeitswerte werden in der Variable `history` gespeichert. So können sie danach (etwa mithilfe von matplotlib[15]) visualisiert werden.

Beim Ausführen dieses Quellcodes mit lediglich 20 000 Trainingsbeispielen kann folgender Graph entnommen werden:

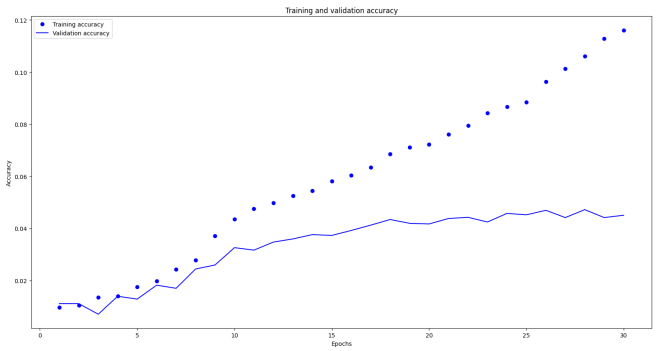

Abb. 6: Lernverlauf eines Trainings mit wenigen Trainingsdaten, erstellt mit matplotlib.

Die x-Achse zeigt hier die Epoche des Trainings, die y-Achse die Genauigkeit des Modells (1,0 würde hier bedeuten, dass das Netzwerk in jedem Fall einer Stellung den gespielten Zug zuordnen kann). Die Punkte repräsentieren die Genauigkeit des Modells bei den Trainingsdaten, die durchgezogene Linie die bei den Validierungsdaten. Es gibt zwar immer ein gewisses statistisches Rauschen, aber dennoch ist eindeutig zu erkennen, dass sich die beiden Graphen etwa ab Epoche 10 trennen. Weil die selben Trainingsdaten mehrfach fürs Training genutzt werden, setzt hier die Überanpassung ein und das Modell wird nicht darin besser, generellen Schachpositionen ihre gespielten Züge zuzuordnen; sondern nur noch, sich sonstig präsente Muster, die nur in den Trainingsdaten vorliegen, einzuprägen.

Falls es sich hierbei um meine einzigen Trainingsdaten handelte, müsste man nun anfangen, das Netzwerk bewusst umzugestalten, um diese Art von Überanpassung zu verhindern. Da durch die Lichess-Datenbank aber viel mehr Daten vorliegen und diese auch noch besser aufbereiten werden können, ist es möglich, Überanpassung völlig außen vor zu lassen, da *kein Trainingsbeispiel doppelt genutzt wird*. Dies wird mit einer Generatorfunktion ermöglicht, die Daten nur dann lädt, wenn sie gerade gebraucht werden.

So kann ich ein Modell trainieren, das nicht überangepasst ist und das bei den Validierungsdaten eine Korrektklassifizierungsrate von etwa 16% erreichen kann. Der Lernverlauf sieht wie folgt aus:

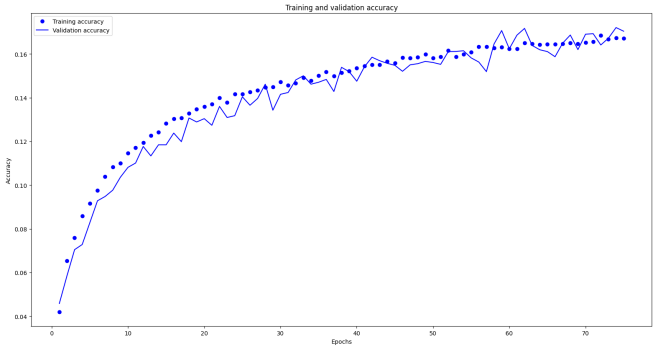

Abb. 7: Lernverlauf eines Trainings mit vielen Trainingsdaten, erstellt mit matplotlib

Durch die Optimierung der Hyperparameter (etwa mehr Schichten, andere Stapelgröße, mehr Neuronen pro Schicht) kann die Korrektklassifizierungsrate des Modells bei bisher unbetrachteten Stellungen auf 20% erhöht werden.

4.5. Wie kommt das fertige Netzwerk in den Browser?

Um mehr Einstellungsmöglichkeiten als bei meinem letzten Projekt zu haben und Zustände wie Einstellungen oder die aktuelle Brettstellung besser handhaben zu können, nutze ich das progressive JavaScript-Framework Vue.js[23]. Das Netzwerk selbst kann nach einem unkomplizierten Konvertierungsprozess mithilfe von tensorflow.js (JavaScript-Port der Tensorflow[22]-Bibliothek) im Browser genutzt werden.

Jetzt müssen die Ein- und Ausgaben in das neuronale Netzwerk nur noch ver- und entschlüsselt werden können. Dies funktioniert in TypeScript ähnlich wie in Rust:

```
export function completeOutputToMoves(
  output: CompleteOutput,
): MoveWithAct[] {
  let moves = [...output]
    .map((act, i) => [act, [i / 64, i % 64]] as const)
    .map(([act, [from, to]]) => {
      let intToChar = (i: number) => String.fromCharCode(97 + i);
      return {
        act,
        from: intToChar(from % 8) + (Math.floor(from / 8) + 1),
        to: intToChar(to % 8) + (Math.floor(to / 8) + 1),
      };
    });

  return moves.sort((a, b) => b.act - a.act);
}
```

Diese Funktion erstellt aus einer neuronalen Ausgabe (hier des Typs `CompleteOutput`, der die 4096-fache Mehrfachklassifizierung repräsentiert) eine Liste von Zügen

mitsamt ihrer neuronalen Aktivierung. Die Methode entpackt zuerst die den Anfang und das Ende des Zuges repräsentierende Zahl zwischen 0 und 63 (ähnlich wie eine Ziffernaufteilung mit der Basis 64). Aus diesen beiden Zahlen werden dann in der gleichen Art und Weise die Linie und Reihe auf dem Schachbrett entnommen. Es wird ein Array erstellt, der alle Züge mitsamt ihrer Aktivierung speichert und dieser wird anschließend, nach der Aktivierung sortiert, zurückgegeben.

Die Methode zum Enkodieren einer Schachstellung als neuronale Eingabe wurde hier der Kürze halber ausgespart, da sie sehr ähnlich zur Rust-Methode `move_to_input` (siehe 4.3) ist.

Mithilfe dieser Methoden und der chess.js[24]-Bibliothek kann man nun automatisch aus den computerlesbaren UCI-Zügen menschenlesbare SAN-Züge erzeugen und diese darstellen. Damit man tatsächlich Schach spielen kann, braucht man allerdings noch ein Schachbrett. In bisherigen Projekten war chessboard.js[25] meine Wahl, jedoch hat sich dieses als für mobile Geräte ungeeignet herausgestellt, weswegen stattdessen chessground[7] genutzt wird. Diese Bibliothek ist von Lichess selbst entwickelt und funktioniert sowohl auf Desktop- als auch mobilen Geräten ohne erweiterte Konfiguration.

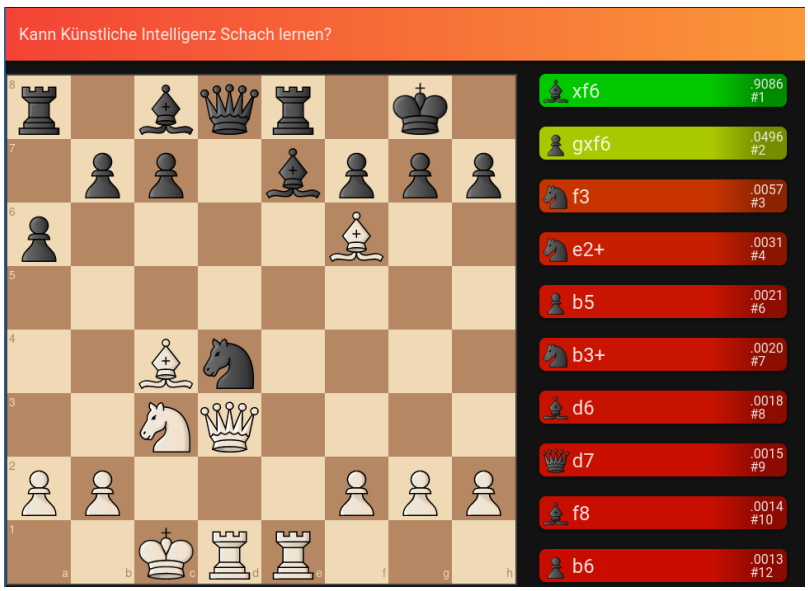

Abb. 8: Dieses Bild zeigt einen Teil der Webanwendung. Links im Bild befindet sich das interaktive Schachbrett (Schwarz ist am Zug), rechts davon die Vorschläge des aktuell geladenen neuronalen Netzwerks mitsamt ihrer Aktivierung. Es ist zu erkennen, dass das Schlagen des Läufers auf f6 eindeutig favorisiert wird - die neuronale Aktivierung ist etwa 0,9, was 90% entspricht.

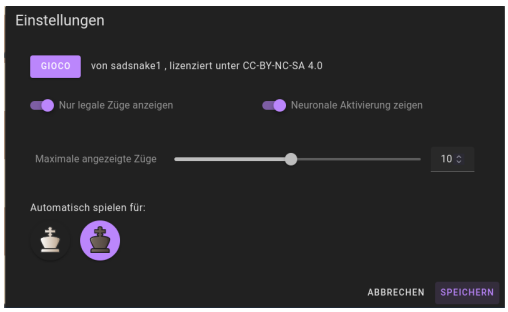

Abb. 9: Die aktuellen Einstellungen, die vorgenommen werden können, sind in diesem Menü vermerkt. Hier können etwa die Spielfiguren angepasst oder Zugdisplayeinstellungen geändert werden. Dazu kann eingestellt werden, dass Weiß oder Schwarz automatisch die Auswahl des neuronalen Netzwerks ausführt – so kann man gegen einen virtuellen Gegner Schach spielen.

5. Fazit

Das Modell, das ich in diesem Projekt entwickelt und trainiert habe, hat eine Korrektklassifizierungsrate von 20% - in dieser Anzahl von Fällen kann der Zug eines Menschen richtig vorhersagt haben. Das mag zuerst nicht besonders gut klingen, aber es ist ein enormer Fortschritt gegenüber einem untrainierten Modell, dessen Rate bei etwa 0,049% läge. Wenn man gegen das Modell Schach spielt, merkt man, dass es menschlicher spielt als eine algorithmische Spielbaumsuche wie Minimax - im Gegensatz zu diesem kann es Figuren in der Eröffnung sinnvoll entwickeln und sich taktische Muster wie Fesselungen oder Gabeln einprägen und sie in einer Spielsituation ausführen.

Überrascht war ich davon, wie gut das Netzwerk legale von illegalen Zügen unterscheiden kann. Der priorisierte Zug ist in mehr als 80% der Fälle legal, also nach den Schachregeln erlaubt – dies ist besonders bemerkenswert, wenn man weiß, dass es meistens nicht mehr als 25 mögliche Züge für eine Farbe gibt. Der Grund für das Vorschlagen eines illegalen Zugs ist meist für einen Menschen nachzuvollziehen, so werden häufig Schachs aus einen längeren Entfernung ignoriert oder ein populärer Zug vorgeschlagen, der nicht möglich ist, weil das Zielfeld von einer eigenen Figur blockiert wird.

Allerdings hat das Modell auch noch einige Fehler und Schwachstellen. Ein Beispiel dafür ist, dass ein Schachmatt sehr selten bis nie vorgeschlagen und ausgeführt wird. Es ist einfach, zu erkennen, warum das der Fall ist: in den ausgewählten Trainingspartien fortgeschrittener Schachspieler werden die meisten Spiele nicht durch ein Schachmatt, sondern durch Aufgabe beendet. So entstehen kaum Trainingsdaten, die zeigen, wie das Mattsetzen funktioniert, geschweige denn dessen Spielvorteil erklären. Außerdem passieren insbesondere im Mittelspiel grobe Patzer wie ein einzügiger Damenverlust (die Dame ist die wertvollste Figur im Schach). Diese sind dadurch zu erklären, dass für diese Situationen die Traininsdaten nicht gut genug abstrahiert werden können, keine Analyse basierend auf dem Wert der Figuren stattfindet und keine Beispiele dafür vorliegen, dass eine solche Figur hier geschlagen werden könnte.

Interessant ist hierbei, dass diese Schwachstelle auch bei heute populären Beispielen künstlicher Intelligenz, insbesondere Textverarbeitungsmodellen, vorliegt – wenn ihre Trainingsdaten etwa sachlich falsche oder diskriminierende Textstellen enthalten, kann das finale Modell auch solchen Text ausgeben. Im Schach kann man dieses Phänomen auf eine harmlose Art und Weise testen und Möglichkeiten ausprobieren, es zu minimieren.

In Zukunft möchte ich das Modell weiterentwickeln und in der Webanwendung mehr Funktionen hinzuzufügen. Zum ersten Punkt gehört, ein Modell basierend auf Taktikrätseln zu trainieren und testen, wie sich dieses gegen einen Menschen schlägt. Außerdem kann ich hybride Ansätze zwischen KI und klassischen Algorithmen genauer betrachten, die diese aktuell die größte Spielstärke aufweisen. In der Anwendung selbst kann ich eine Visualisierung des Modells sowie mehr Spieloptionen einfügen, etwa einen Temperaturparameter, mithilfe dessen nicht immer den aktivierteste Zug, sondern mit einem gewichteten Zufall ein anderer gespielt wird – so erhalten die Spiele mehr Vielfalt und Varietät.

Literatur und Quellenangabe

[1] Duplessis, T., & others. Lichess – "Kostenloses Online-Schach" – https://lichess.org (zuletzt abgerufen: 20.11.2022)

[2] Blume, L. Effizienzanalyse des Minimax-Algorithmus im Bezug auf Schach

[3] Fatir, A. Kernel Image Processing : Image Filters (with Java Code) – http://tech.abdulfatir.com/2014/05/kernel-image-processing.html (zuletzt abgerufen: 06.01.2023)

[4] Khlestov, I. Different types of the convolution layers – https://ikhlestov.github.io/pages/machine-learning/convolutions-types/ (zuletzt abgerufen: 12.01.2022)

[5] Chevalier, G. The LSTM Cell, under CC-BY-SA 4.0

[6] Rodriguez, J. The Science behind OpenAI Five that just Produced One of the Greatest Breakthroughs in the History of AI – https://web.archive.org/web/20191226222000/https://towardsdatascience.com/the-science-behind-openai-five-that-just-produced-one-of-the-greatest-breakthrough-in-the-history-b045bcdc2b69?gi=24b20ef8ca3f (archiviert am 26.12.2019, Archiv zuletzt abgerufen: 06.01.2023)

[7] Duplessis, T., & others. chessground – Mobile / Web chess UI for lichess.org

[8] Nielsen, M. Neural networks and deep learning – http://neuralnetworksanddeeplearning.com/ (zuletzt abgerufen: 04.01.2023)

[9] MV, P. Top 10 Reasons Why 87% of Machine Learning Projekts Fail – https://dzone.com/articles/top-10-reasons-why-87-of-the-machine-learning-proj (zuletzt abgerufen: 04.01.2023)

[10] Datta, P. Why Do Most AI Projects Fail? – https://www.forbes.com/sites/forbestechcouncil/2020/10/14/why-do-most-ai-projects-fail/ (zuletzt abgerufen 05.01.2023)

[11] Chollet, F. Deep Learning mit Python und Keras. mtip

[12] Vignesh, S. The Perfect Fit for a DNN – https://medium.com/analytics-vidhya/the-perfect-fit-for-a-dnn-596954c9ea39 (zuletzt abgerufen: 07.01.2023)

[13] Yathish, V. Loss Functions and Their Use in Neural Networks – https://towardsdatascience.com/loss-functions-and-their-use-in-neural-networks-a470e703f1e9 (zuletzt abgerufen: 13.01.2023)

[14] van Rossum, G. The Python Programming Language – https://www.python.org/ (zuletzt abgerufen: 12.01.2023)

[15] Hunter, J. Matplotlib: Visualization with Python – https://matplotlib.org/ (zuletzt abgerufen: 10.01.2023)

[16] Wood, T. Convolutional Neural Network – https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network (zuletzt abgerufen: 10.01.2023)

[17] Duplessis, T. lichess.org open database – https://database.lichess.org/ (zuletzt abgerufen: 06.01.2023)

[18] Creative Commons CC0 license – https://creativecommons.org/choose/zero/ (zuletzt abgerufen: 06.01.2023)

[19] Specht, P. Die 50 wichtisten Themen der Digitalisierung. Redline Verlag 4. Auflage 2019. (S. 181)

[20] Crowther, M. The Week in Chess – https://theweekinchess.com/twic (zuletzt abgerufen: 06.01.2023)

[21] Chollet, F., & others. Keras: the deep learning API – https://keras.io (zuletzt abgerufen: 18.12.2022)

[22] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heretogeneous systems. – https://www.tensorflow.org

[23] Evan, Y., & others. Vue.js – The Progressive JavaScript Framework – https://vuejs.org (zuletzt abgerufen: 07.01.2023)

[24] Hlywa, J., & others. chess.js – A JavaScript chess library for chess move generation/validation, piece placement/movement, and check/checkmate/draw detection – https://github.com/jhlywa/chess.js (zuletzt abgerufen: 11.01.2023)

[25] Oakman, C., & others. chessboard.js – The easiest way to embed a chess board on your site. (zuletzt abgerufen: 11.01.2023)

[26] Miszczak, P. GPT-3 Statistics: Usage, Parameters, Use Cases & More - https://businessolution.org/gpt-3-statistics/ (zuletzt abgerufen: 13.01.2023)

[27] Peyrard, Clément & Mamalet, Franck & Garcia, Christophe. (2015). A Comparison between Multi-Layer Perceptrons and Convolutional Neural Networks for Text Image Super-Resolution. VISAPP 2015 - 10th International Conference on Computer Vision Theory and Applications; VISIGRAPP, Proceedings.

[28] Chollet, F. Deep Learning mit Python und Keras. mtip (S. 345-349)

[29] Hoare, G., & others. Rust: A language empowering everyone to build reliable and efficient software. (zuletzt abgerufen: 12.01.2023)

[30] Ertel, Wolfgang. (2020). Künstliche Intelligenz -Was ist das?. Februar 2020.