HAW HAMBURG

BACHELOR THESIS
Kjell May

Entwicklung eines bildbasierten Assistenten für chinesisches Schach

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Kjell May

Entwicklung eines bildbasierten Assistenten für chinesisches Schach

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Peer Stelldinger
Zweitgutachter: Prof. Dr. Marina Tropmann-Frick

Eingereicht am: 01.08.2023

Kjell May

Thema der Arbeit

Entwicklung eines bildbasierten Assistenten für chinesisches Schach

Stichworte

Schach, Chinesisches Schach, XiangQi, Bilderkennung, Adaptive Thresholding, Contour Finding, Curve Approximation, Machine Learning, ImageNet, Focal loss, FEN, Schachengine

Kurzzusammenfassung

In dieser Arbeit wird ein Algorithmus vorgestellt, der auf Basis eines Bildes von einem chinesischen Schachbrett dessen Stellung berechnet und an eine Engine weitergibt, welche den besten Zug zurückgibt. Das Brett wird dabei mittels adaptivem Thresholding und Contour Finding erkannt. Die Figuren werden mithilfe eines neuronalen Netzes auf Basis von ImageNet klassifiziert und zur vorliegenden Stellung in Form eines FEN-Strings zusammengebaut. Dieser wird an die Engine Pikafish gegeben, um den besten Zug für die Seite, aus der das Bild gemacht wurde, über die Konsole auszugeben. Bei Bildern aus der top-down-Perspektive oder aus einem kleinen Winkel wird das Brett in über 95% und die Stellung in über 60% der Fälle korrekt erkannt. Unter optimalen Bedingungen ist die Erfolgsrate noch höher. Für Bilder aus einem flachen Winkel zum Brett sowie bei Bildern mit Gegenlicht oder Lichtflecken ist der Algorithmus jedoch noch nicht robust genug und erreicht eine niedrige Genauigkeit.

Kjell May

Title of Thesis

Development of an image-based assistant for Chinese chess

Keywords

Chess, Chinese chess, Xiangqi, Image recognition, Adaptive Thresholding, Contour Finding, Curve Approximation, Machine Learning, ImageNet, Focal loss FEN, chess engine

Abstract

In this thesis, an algorithm is presented which, on the basis of an image of a Chinese chessboard, calculates its position and passes it on to an engine, which returns the best move. The board is detected by means of adaptive thresholding and contour finding. The pieces are classified with the help of a neural network based on ImageNet and assembled to represent the position in the form of a FEN string. This is given to the engine Pikafish to output the best move via the console for the side from which the image was taken from. For images taken from the top-down perspective or from a slight angle, the board is correctly recognised more than 95% of the time, the position in more than 60%. For images taken from a low angle to the board and for images with backlighting or light spots, however, the algorithm is not yet robust enough and achieves low accuracy.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Schach ist eines der ältesten, weltweit verbreitetesten Brettspiele der Welt. In der westlichen Welt hat es eine tiefe kulturelle Bedeutung erlangt und wurde bereits als Sportart anerkannt. Seit Jahrhunderten wird dieses Spiel immer tiefgründiger analysiert, wobei seit Ende des 20. Jahrhunderts auch Computer vermehrt als Unterstützung zum Einsatz kommen. Etabliert haben sich solche Computer 1996, als erstmalig ein Schachcomputer von IBM names *DeepBlue* einen menschlichen Schachweltmeister schlagen konnte [35].

Eher unbekannt in der westlichen Welt, aber umso vertiefter in der chinesischen Kultur, ist das chinesische Schach *XiangQi* (象棋 übersetzt etwa "Elefantenspiel").
XiangQi besitzt einige Ähnlichkeiten zum westlichen Schach (siehe Kapitel 3) mit einer ähnlichen Zustandsraumkomplexität (10^{40} zu 10^{50})[32], einer jedoch signifikant größeren Spielbaumkomplexität von 10^{150} (Schach: 10^{123})[8].

Mit den heutzutage fortgeschritteneren Computern und Möglichkeiten wurde auch begonnen, Schach mit Bilderkennung/- verarbeitung zu betrachten. Dabei wurde schon mit unterschiedlichsten Methoden herangegangen, einige Beispiele werden in Kapitel 2 besprochen. Zu XiangQi lässt sich jedoch wenig finden, bzw. nicht an den Orten, an denen üblicherweise für uns zugänglich wissenschaftliche Arbeiten veröffentlicht werden. Es kann sicherlich trotzdem davon ausgegangen werden, dass sich in China auch damit beschäftigt wird, es nur schwieriger zu finden ist.

Unter anderem deswegen wurde sich in dieser Arbeit für dieses Thema entschieden. Ich habe außerdem einen persönlichen Bezug zu dem Spiel. Es wurde mir in der 9. Klasse der Schule von einem Lehrer für chinesisches Schach beigebracht und ich habe ein Jahr später bei der "1st CAISSA Youth World Championship" 2016 in Hamburg teilgenommen. Dort konnte ich den 11. Platz in meiner Altersklasse allgemein und den 3.Platz der "nicht-chinesisch-nicht-vietnamesisch"-Wertung erreichen [2][19].

Der Idee des "bildbasierten Assistenten" für diese Arbeit liegt zugrunde, dass ein*e Spieler*in zu einem beliebigen Zeitpunkt während eines Spiels ein Bild vom Brett machen

kann, um dann einen Zugvorschlag zu erhalten. Dies soll natürlich kein unfaires Verhalten oder Betrügen unterstützen, sondern vielmehr den Aufwand eines manuellen Übertragens einer Stellung ins Digitale zu einer Engine reduzieren.

In den folgenden Kapiteln wird zunächst *Related Work* (Forschungsstand/Verwandte Arbeiten) präsentiert, grundlegend die Regeln des chinesischen Schachs erläutert und die Unterschiede zum westlichen Schach aufgezeigt und anschließend das Vorgehen für diese Arbeit erläutert. Dann wird der Algorithmus in seinen Einzelschritten dargestellt, um auf Basis der Experimente ein Fazit zu ziehen und Ausblicke zu geben. In dieser Arbeit werden Fachbegriffe und englische Begriffe, die erstmalig auftreten, kursiv geschrieben, um sie dann in normaler Schreibweise zu verwenden.

2 Related Work

Wie bereits in der Einleitung erwähnt, wurde im Bereich der Bildverarbeitung und Objekterkennung für westliches Schach schon vieles untersucht. Orémuš, Z. hat in seiner Masterarbeit von 2018 einen Algorithmus entwickelt, mit dem versucht wird das Schachbrett mithilfe von *Canny Edge Detector* und *Hough Transformation* zu erkennen und die Figuren zu klassifizieren, wie es auch Kafafy, M. und Danner, C. vor ihm untersucht haben [10][30]. In seiner Thesis referenziert er Gupta, S. und Varun, R., welche zusammen einen sehr ähnlichen Ansatz hatten, jedoch nutzen diese ein *R-CNN (Region Based Convolutional Neural Network)*, während Orémuš, Z. die Klassifizierung mittels *Template Matching* durchführt. Andere Ansätze zum Identifizieren des Bretts sind das Finden von Eckpunkten mit *Harris Corner Detector* und *SIFT Descriptor Classifier*, wie es Hack, J. und Ramakrishnan, P. in [17] angehen. Auch die Vereinfachung durch minimale Benutzerinteraktion für das Festlegen der Eckpunkte des Bretts, ist ein Ansatz von Ding, J. (siehe [11]).

Da sich ein Schachbrett aufgrund seines Aufbaus mit abwechselnd schwarzen und weißen Feldern sehr gut für die Kalibrierung von Kameras anbietet, stellt OpenCV[1] - eine Bibliothek mit Algorithmen im Bereich der Bildverarbeitung - die Methode `cv.findChessboardCorners()` zur Verfügung, die genau darauf aufbaut.

Wie sich erkennen lässt, wurde dieses Themengebiet schon sehr ausführlich untersucht. Jedoch lässt sich wenig im Bezug auf chinesisches Schach ausfindig machen. Ein vorliegendes Paper (siehe [18]) ist auf Chinesisch verfasst und musste mithilfe von Übersetzungsprogrammen nachvollzogen werden. In dem Paper wird *Robert Edge Detection* und *Hough Circle Transformation* genutzt, um anschließend durch *Histogram Projection* und *Fast Fourier Transform* die Figuren unabhängig von der Rotation zu klassifizieren. Dies wird durchgeführt mit Bildern der Figuren aus der Vogelperspektive auf einem flachen weißen Spielfeld, es stellt also einen vereinfachten Ansatz dar.

[1] https://opencv.org, Zugriff Juli 2023

Das Ziel in dieser Arbeit war es jedoch, die Stellung aus unterschiedlichen Winkeln für ein dreidimensionales Brett zu erkennen. Aus diesem Grund lässt sich wenig aus dem vorher genannten Paper übertragen.

3 XiangQi-Regeln und Unterschiede zum westlichen Schach

Grundlegend unterscheiden sich Schach und XiangQi nicht allzu sehr. Es gibt ein Brett mit Figuren, wovon eine der König bzw. im XiangQi der General ist, welcher Schachmatt gesetzt werden muss, um zu gewinnen. Ein wesentlicher Unterschied, unabhängig von den Regeln, ist, dass die Figuren nicht auf die Felder, sondern auf die Schnittpunkte dieser gesetzt werden. Die Figuren im XiangQi haben außerdem alle die gleiche Form, vergleichbar mit denen aus Backgammon oder Dame, und sind unterscheidbar durch ihr chinesisches Symbol. Es gibt jedoch auch Bretter mit Symbolen für nicht-chinesische Spieler (siehe Abb. 3.1). Folgend werden kurz das Brett und die Figuren erklärt. Tiefergehende Regeln sind hier nicht weiter relevant. Alle Bilder in diesem Kapitel stammen aus [3].

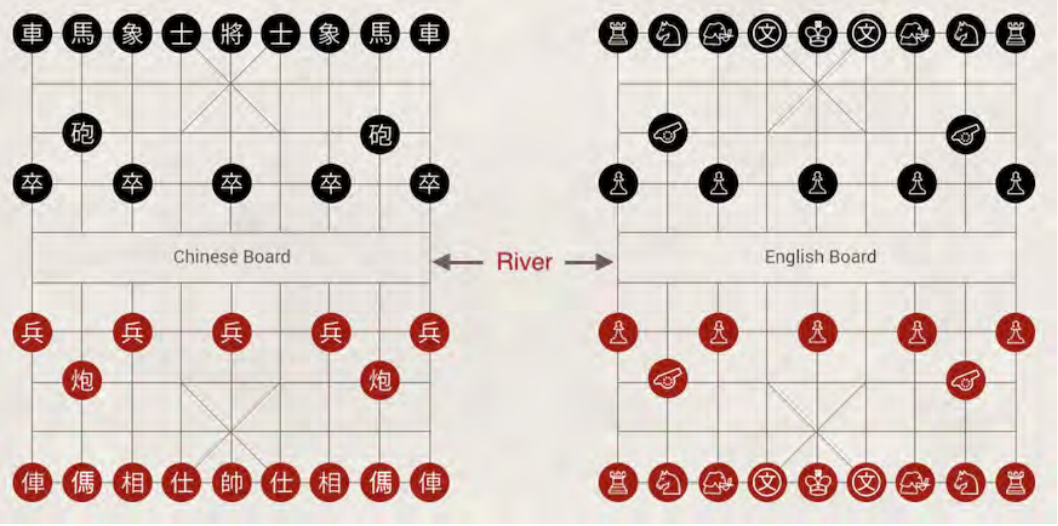


Abbildung 3.1: Startaufstellung im XiangQi

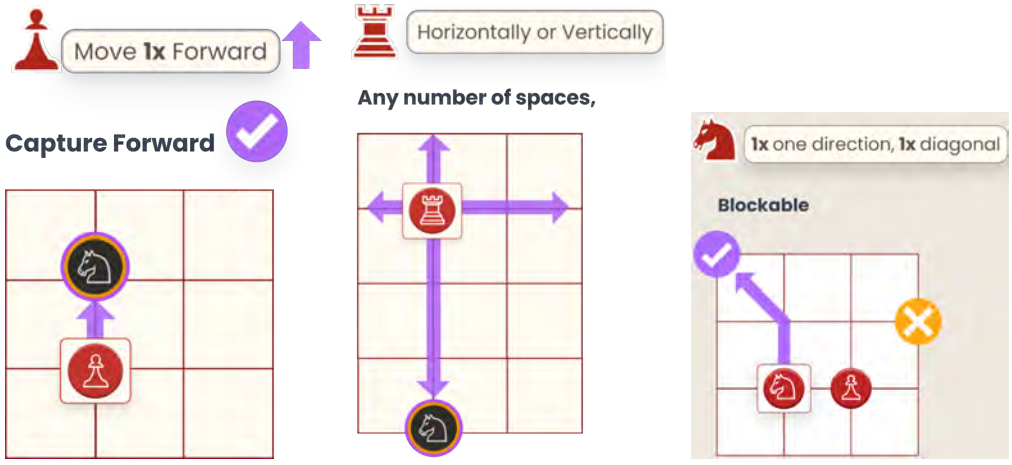
3.1 Spielbrett

Das Spielbrett im chinesischen Schach ist 9x10 Felder groß, mit einem sogenannten "Fluss" in der Mitte und jeweils sogenannten "Palästen" hinten mittig bzw. vorne mittig. Der Palast ist der Bereich, in dem sich der General und seine Berater bewegen dürfen. Den Fluss dürfen einige Figuren nicht überqueren, dazu in der nächsten Sektion Genaueres.

3.2 Figuren

Die Figuren aufgelistet mit jeweils vergleichbaren Figuren im westlichen Schach und wie sie sich bewegen und schlagen dürfen:

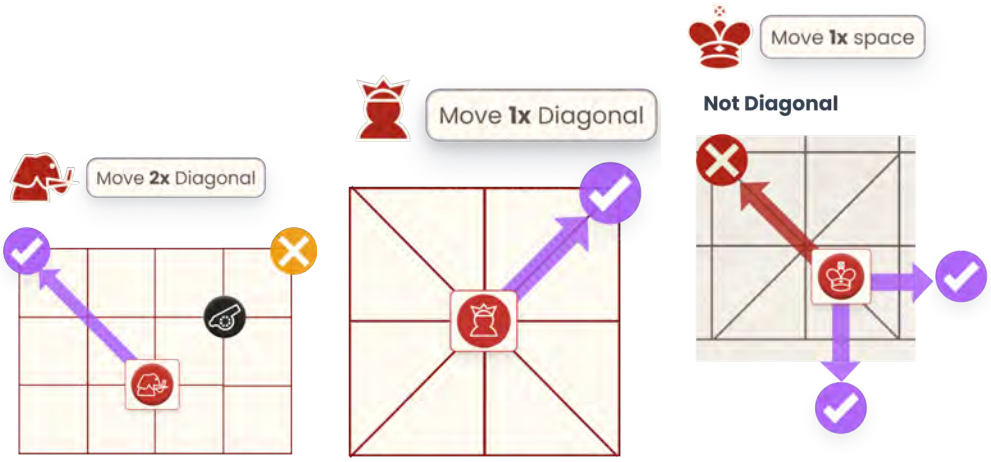
- Soldat (Abb. 3.2a) (≙ Bauer). Bewegung und Schlagen geradeaus. Nachdem der Fluss überquert wurde, auch beides seitwärts erlaubt.
- Streitwagen (Abb. 3.2b) (≙ Turm). Bewegung und Schlagen genau wie ein Turm im westlichen Schach.
- Pferd (Abb. 3.2c) (≙ Pferd). Bewegung und Schlagen wie ein Pferd im Schach, kann jedoch nicht über Figuren springen, ist also blockbar.



(a) (b) (c)

Abbildung 3.2: Bewegung (a) Soldat (b) Streitwagen (c) Pferd

- Elefant (Abb. 3.3a) (≙ Läufer). Bewegung und Schlagen jeweils zweimal schräg in eine Richtung, ohne Figuren zu überspringen und kann Fluss nicht überqueren.
- Berater (Abb. 3.3b) (nichts vergleichbar im Schach). Bewegung und Schlagen nur innerhalb des Palastes auf den schrägen Linien.
- General (Abb. 3.3c) (≙ König). Bewegung und Schlagen nur innerhalb des Palastes auf den geraden Linien.



(a) (b) (c)

Abbildung 3.3: Bewegung (a) Elefant (b) Berater (c) General

- Kanone (Abb. 3.4) (nichts vergleichbar im Schach). Bewegung wie ein Turm. Zum Schlagen muss eine Figur dazwischen stehen.

Abbildung 3.4: Bewegung Kanone

4 Vorgehen

Da es sich angeboten hat, im Rahmen dieser Arbeit ein eigenes Spielbrett zu nutzen und die Verfügbarkeit von Datensätzen für XiangQi sehr niedrig ist, wurde mit diesem ein eigener Datensatz erstellt. Dafür wurde das Brett auf einem Tisch aufgebaut und Bilder aus verschiedenen Winkeln (*top-down*, leicht angewinkelt und flacher Winkel zum Tisch, siehe Abb. 4.1) aufgenommen. Dies wurde jeweils von beiden Seiten des Tisches durchgeführt, wobei zur Variation der Beleuchtung aus einer Ansicht eine Lichtquelle im Rücken und bei der anderen gegenüber liegt. Anschließend wurde die Stellung etwas verändert und der Vorgang wiederholt, womit ein Datensatz der Größe von ca. 200 Bildern erreicht wurde. Dieses Vorgehen sollte in einem möglichst diversen Datensatz resultieren.
Dieser erste Satz an Bildern wurde dann genutzt, um jeweils die Spielbretter zu erkennen und zu extrahieren, also das Bild so zu entzerren und zuzuschneiden, dass es nur noch aus dem Spielbrett besteht. Diese Bilder wurden dann weiter genutzt, um die Figuren einzeln auszuschneiden, welche dann für das Training eines neuronalen Netzes zur Klassifikation dieser dienten.
Alle Bilder wurden mit einem "iPhone SE 2020" aufgenommen, programmiert wurde ausschließlich in Python 3.10. Genaueres zur jeweiligen Vorgehensweise wird in den zugehörigen Unterkapiteln im Kapitel 5 beschrieben.

Abbildung 4.1: Startaufstellung aus drei Winkeln

5 Algorithmus

Im Folgenden wird nun der Ablauf des Algorithmus vorgestellt, unterteilt in fünf wesentliche Schritte, die jeweils ein Kapitel bilden:

1. Im ersten Schritt wird das Spielbrett erkannt und entzerrt (Kapitel 5.1).
2. Im zweiten Schritt wird das Brett in gleich große Ausschnitte aufgeteilt, sodass jeder ein einzelnes Feld abbildet (Kapitel 5.2).
3. Im dritten Schritt werden diese Bilder mittels eines neuronalen Netzes klassifiziert (Kapitel 5.3).
4. Im vierten Schritt wird auf Basis der Klassifikationen die Stellung für die Engine verständlich zusammengebaut (Kapitel 5.4).
5. Im fünften Schritt wird die ermittelte Stellung an eine Engine gegeben, welche einen Zugvorschlag ausgibt (Kapitel 5.5).

Für das Arbeiten mit den Bildern werden die Bibliotheken `OpenCV` (Version 4.7.0) und `numpy` (Version 1.23.5) genutzt.

5.1 Bretterkennung

In diesem Kapitel werden verschiedene Ansätze zur Bretterkennung inklusive ihrer Anwendung vorgestellt (5.1.1, 5.1.2, 5.1.3), wie mit dem erkannten Brett fortgefahren wird (5.1.4) und wie auftretende Fehler behandelt werden (5.1.5).
Um Bilder schneller verarbeiten zu können, wird das Brett mittels `cv.resize()` herunter skaliert auf eine Größe von 806x605 Pixeln, welche ein Fünftel der Originalgröße darstellt (4032x3024 bei der genutzten Kamera).

5.1.1 Ansatz Canny Edge Detector und Hough Line Transformation

Der erste Ansatz war, das Brett durch Canny Edge Detector und Hough Transformation zu erkennen, so wie es in [10] und [30] gemacht wurde.

Canny Edge Detector

Der Canny Edge Detector versucht Kanten in einem Bild zu finden. Dies soll dazu dienen, die Menge an Daten zu reduzieren, nutzlose Informationen herauszufiltern, dabei aber wichtige strukturelle Eigenschaften beizubehalten [16]. Entwickelt wurde dieser Algorithmus von John Canny (siehe [9]), in OpenCV ist dieser implementiert als `cv.Canny()`.
Der Detector duchläuft mehrere Schritte (aus [26]):

1. Um Rauschen zu entfernen und das Bild zu glätten, wird ein 5x5 *Gaussscher Filter* auf das Bild angewendet.

2. Anschließend wird das Bild mit einem *Sobel Kernel* gefiltert, um den *Intensitäts-Gradienten* an jedem Punkt (x, y), sowie seine Richtung zu finden. Der Gradient berechnet sich durch: $G = \sqrt{{G_x}^2 + {G_y}^2}$, die Richtung durch: $\theta = \tan^{-1}(\frac{G_y}{G_x})$

3. Die Richtung des Gradienten ist immer senkrecht zu den Kanten, also muss θ in eine Richtung gerundet werden. Abb. 5.1 zeigt Rundungen für Winkelbereiche. Der gelbe Bereich wird dabei auf 0° gerundet.

4. Dann wird *Non-maximum suppression* durchgeführt, das heißt, das gesamte Bild wird gescannt und alle Punkte, die nicht zu einer Kante gehören, werden entfernt. Übrig bleiben dann dünne Linien, welche die Kanten darstellen.

5. Im letzten Schritt (genannt *Hysteresis*) werden alle Kanten durchgegangen und bestimmt, ob sie tatsächlich eine Kante darstellen. Dafür werden zwei *Thresholds* (Schwellenwerte) T1 und T2 für Gradientenwerte bestimmt. Liegt der Wert einer Kante über T2, dann ist es sicher eine Kante. Liegt er unter T1, ist es keine Kante. Im Falle, dass er dazwischen liegt, wird geschaut, ob die Kante zu einer bereits "sicheren" Kante gehört. Falls dies zutrifft, wird diese auch als Kante festgelegt, sonst verworfen.

Auf ein Bild, welches nur die Struktur des Bildes (also die Kanten) zeigt, lässt sich anschließend gut die Hough Transformation anwenden.

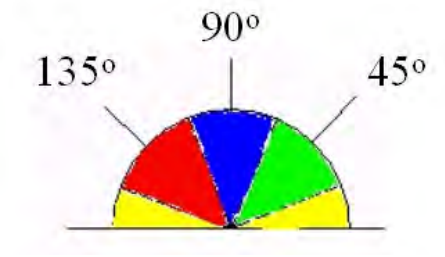


Abbildung 5.1: Rundung von θ (aus [16])

Hough Transformation

Die Hough Transformation wird unter anderem dafür genutzt, um Linien in einem Bild zu finden. Für Hough Transformationen werden Linien im Polarkoordinatensystem statt im Kartesischen Koordinatensystem ausgedrückt. Das bedeutet, eine Gerade der Form $y = mx+b$ wird umgewandelt in $y = (-\frac{\cos\theta}{\sin\theta})x+(\frac{r}{\sin\theta})$, umgestellt nach r: $r = x\cos\theta+y\sin\theta$ (Bildliche Darstellung: Abb. 5.2). Die Unbekannten sind also r und θ. Werden jetzt verschiedene (x, y) Paare in der Ebene geplottet, sodass $r > 0$ und $0 < \theta < 2\pi$, resultiert dies jeweils in Sinuskurven. Schneiden sich die Kurven von zwei verschiedenen Punkten, gehören sie zu einer Linie, die durch den Schnittpunkt (r, θ) definiert ist (Beispiel siehe 5.3). Festgelegt werden kann dann, wie viele Schnittpunkte mindestens benötigt werden, um eine Linie zu ermitteln [28]. In OpenCV ist dieser Algorithmus implementiert als `cv.HoughLines()`.
Eine effizientere robustere Variante der Hough Transformation ist die *Progressive Probabilistic Hough Transformation (PPHT)*. Dabei wird die Berechnungszeit reduziert, indem "der Unterschied der Anteile der Stimmen von Stützpunkten, die erforderlich sind, um zuverlässig Linien mit einer unterschiedlichen Anzahl von Stützpunkten zu erkennen, ausgenutzt wird" (übersetzt aus [22]). Das bedeutet, dass nach dem Prinzip für Wahlalgorithmen lange Linien einen geringeren Anteil von "Stimmen" der Stützpunkte benötigen als kurze Linien, um auf eine tatsächliche Linie im Bild zu schließen. Implementiert in OpenCV ist dies als die Methode `cv.HoughLinesP()`, welche die Extrema der ermittelten Linien (x_0, y_0, x_1, y_1) zurückgibt.

Anwendung auf Bilder des Datensatzes

Mit den erwähnten Methoden von OpenCV lassen sich die Algorithmen einfach auf Bilder des Datensatzes anwenden. Nach Testen verschiedenster Parameter stellten sich $T_1 = 135$

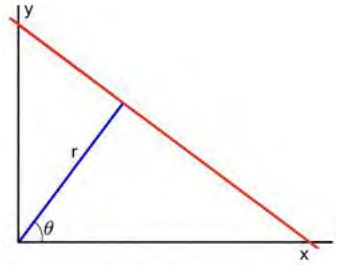


Abbildung 5.2: Bezug der Parameter des kartesischen und Polarkoordinatensystems (aus [28])

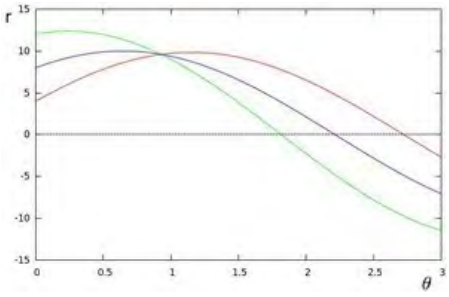
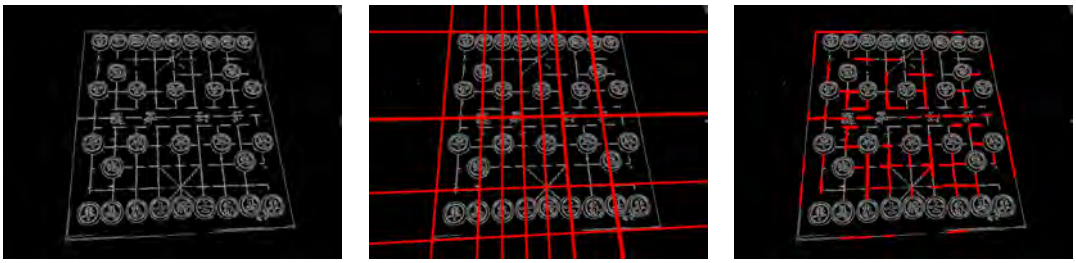


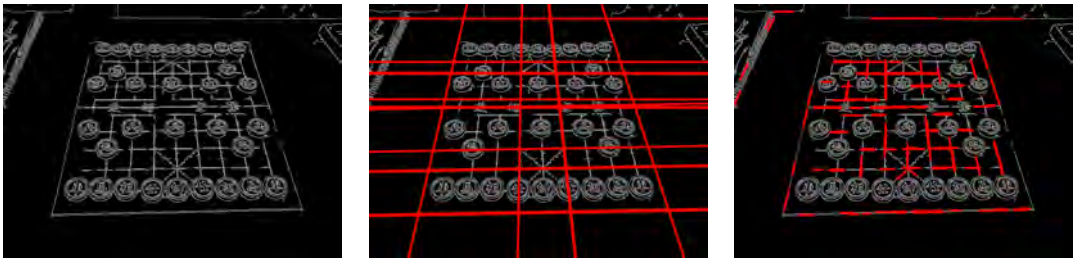
Abbildung 5.3: Sinuskurven drei verschiedener Punkte in der (r, θ) Ebene (aus [28])

und $T_2 = 250$ (für Canny) und $\rho = 1, \theta = \frac{\pi}{180}$ und $threshold = 180$ (für Hough Lines) für das Erreichen der besten Ergebnisse heraus. Diese Ergebnisse sind jedoch nicht gut genug, um damit weiterzuarbeiten. Es werden nicht immer alle Linien des Bretts erkannt, auch nicht zuverlässig wenigstens die Umrisse des Bretts. Da das genutzte Brett ein Holzbrett ist, werden bei niedrigeren Thresholds die Maserungen erkannt, was also auch keine Verbesserung wäre. Auch Versuche mit dem probabilistischen Hough Transform liefern keine Verbesserung. Beispielanwendungen für ein Brett aus zwei Perspektiven zu sehen in Abb. 5.4 und 5.5.



(a) (b) (c)

Abbildung 5.4: Anwendung von (a) Canny (b) Hough Transform (c) PPHT auf Brett



(a) (b) (c)

Abbildung 5.5: Anwendung von (a) Canny (b) Hough Transform (c) PPHT auf Brett (flacher Winkel)

5.1.2 Ansatz Kamera-Kalibrierung

Eine Kamera zu kalibrieren bedeutet *tangentiale* und *radiale* Verzerrungen zu korrigieren, indem eine Übersetzung vom realen Koordinatensystem (3D) zum Kamera-Koordinatensystem (2D) stattfindet. Tangentiale Verzerrung lässt Teile des Bildes näher wirken, als sie tatsächlich sind. Radiale Verzerrung sorgt dafür, dass gerade Linien gebogen aussehen. Dies basiert auf dem Lochkamera-Modell [25]. Mathematisch darstellen lassen sich die Verzerrungen wie folgt (aus [25]):

- radial: $x_{distorted} = x(1+k_1r^2+k_2r^4+k_3r^6)$ und $y_{distorted} = y(1+k_1r^2+k_2r^4+k_3r^6)$
- tangential: $x_{distorted} = x + [2p_1xy + p_2(r^2 + 2x^2)]$ und $y_{distorted} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$

Daraus ergeben sich dann 5 Parameter, die gefunden werden müssen, die *distortion coefficients* $= (k_1\ k_2\ p_1\ p_2\ k_3)$. Zusätzlich werden die *intrinsischen* und *extrinsischen* Parameter der Kamera benötigt. Die Matrix für intrinsische Parameter (mit *focal length*/Brennweite f_x, f_y und *optical centers* c_x, c_y) ergibt sich wie folgt:

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

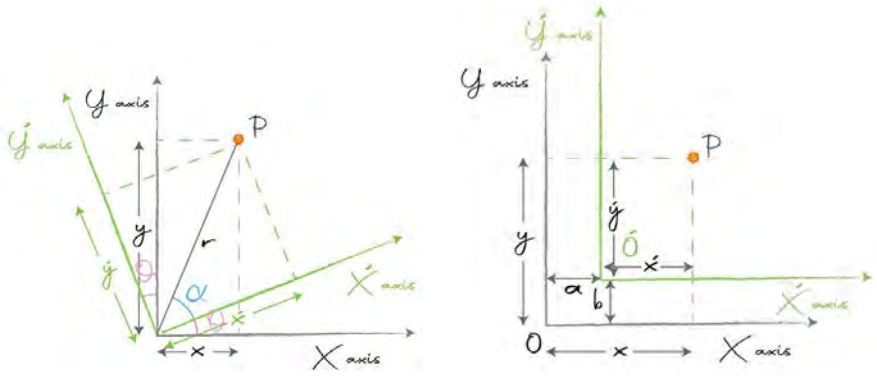
Extrinsische Parameter bestehen aus Rotation und Translation (Übersetzung). Bei der Rotation ist das Ziel die Koordinaten eines Punktes auf Basis einer neuen Basis, das heißt eines rotierten Koordinatensystems, zu finden. Hier wird außerdem angenommen, dass sich auf $Z = 0$ im realen Koordinatensystem befunden wird [20][37] (vgl. Abb. 5.6a).

Die Berechnung für einen Punkt ergibt sich dann als:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & +\cos\theta \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix}$$

Bei der Translation geht es darum einen Punkt um einen Offset (a,b) zu verschieben. Damit die Berechnug mit der Matrix auf den Punkt (x, y) funktionieren kann, wird eine zusätzliche Dimension definiert, deren Koordinaten "homogene Koordinaten" genannt werden (siehe Abb. 5.6b). Die Berechnung sieht dann wie folgt aus:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



(a) (b)

Abbildung 5.6: (a) Rotation, (b) Translation des Koordinatensystems bezogen auf einen Punkt (x, y) (aus [20])

Auch hierfür bietet OpenCV eine Methode an namens `cv.calibrateCamera()`, welche auf Basis von übergebenen *object points* (Objektpunkten) und *image points* (Bildpunkten) die intrinsische Matrix der Kamera, die distortion coefficients und Rotations- und Translationsvektoren berechnet. Aufgebaut ist dieser Algorithmus auf der Methode für Kamerakalibrierung von Zhang, Z., auf der auch die obigen Ansätze basieren [37]. Eine Liste von Bildpunkten (2D Koordinaten im Bild) und ihren zugehörigen Objektpunkten (3D Koordinaten aus der realen Welt) werden hier aufeinander abgebildet. Die Objektpunkte können dabei willkürlich gewählt sein, solange sie zu dem Muster der Bildpunkte passen.

Dieses Muster sollte also möglichst zuverlässig gefunden werden können. Besonders gut bieten sich dafür die Schnittpunkte der Felder eines Schachbretts an. Gesucht werden also genau die Stellen, wo ein Muster der (vereinfachten) Form

0	255
255	0

bzw.

255	0
0	255

auftaucht.

OpenCV stellt dafür mit `cv.findChessboardCorners()` eine Hilfestellung.
Jedoch existieren solche Muster im chinesischen Schach nicht. Zum einen stehen dort die Figuren auf den benötigten Schnittpunkten und zum anderen besitzen alle Felder dieselbe Farbe, es gibt also keine schwarz-weiß Unterscheidung. Nur die vier Eckpunkte eines Bretts als Bildpunkte zu nehmen reicht für eine gute Kalibrierung nicht aus.

5.1.3 Adaptive Thresholding, Contour Finding und Curve Approximation

Der dritte, letztlich erfolgreichste Ansatz beinhaltet *Adaptive Thresholding* (Adaptives Schwellenwertverfahren), *Contour Finding* (Finden von Konturen) und *Curve Approximation* (Kurvenapproximation).

Thresholding

Zuerst wird einfaches Thresholding vorgestellt, um darauf aufbauend Adaptive Thresholding zu erklären:
Ist ein graustufiges Bild gegeben, wird auf jeden Pixel ein *threshold value* angewandt. Je nach Methode wird dieser im nächsten Schritt unterschiedlich neu gesetzt. In der einfachsten Variante (globales binäres Thresholding) werden alle Pixel, die sich oberhalb des Schwellenwerts befinden, auf ein *max value* (meist 255) gesetzt, alle unterhalb auf null [29]. Verschiedene Methoden, wie sie in OpenCV definiert sind, zeigt Abb. 5.7.

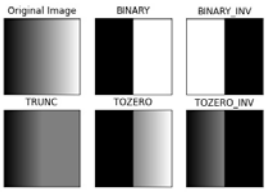


Abbildung 5.7: Threshold Varianten (aus [29])

Jedoch ist globales binäres Thresholding nicht sehr robust. Wenn nichts über das Bild bekannt ist, muss der Wert willkürlich gewählt werden, was aufwändig und schlecht abschätzbar ist.
Eine Lösung für dieses Problem soll *Otsu Binarization* bieten. Anstatt einen konkreten Wert festzulegen, berechnet dieser Algorithmus automatisch den optimalsten. Dabei wird das Histogramm des Bildes analysiert und auf dessen Basis ein optimaler globaler Wert festgelegt, mit dem das Thresholding durchgeführt wird [29]. Eine Beispielanwendung auf ein einfaches Bild mit Rauschen zeigt Abb. 5.8.

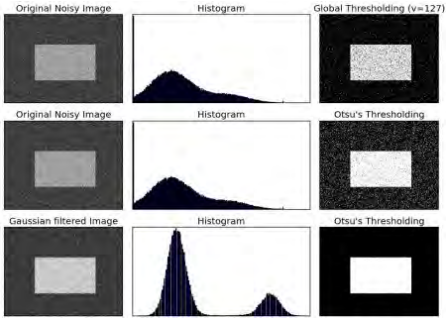


Abbildung 5.8: Otsu Thresholding mit Histogramm-Darstellung (aus [29])

Diese Methode hilft bei der Wahl für einen Schwellenwert, jedoch ist dieser immer noch ein globaler Wert. Eine deutlich robustere Variante gegen unterschiedliche Lichtverhältnisse in verschiedenen Bereichen des Bildes stellt Adaptive Thresholding dar.
Die Idee hierbei ist, dass der Schwellenwert für jeden Pixel auf Basis seiner Nachbarschaft festgelegt wird. Die Pixel der Nachbarschaft können dabei als Durchschnitt oder als gewichtete Summe mit einem *Gaussian Window* (Gausssches Fenster) einfließen [29]. Diese Methode eignet sich für die Anwendung in dieser Arbeit am besten, ist jedoch gleichzeitig am rechenaufwändigsten, da für jeden Pixel einzeln der Schwellenwert berechnet werden muss.

Contour Finding

Um jetzt die Umrisse des Bretts in den Bildern zu finden, werden *Contours* (Konturen) gesucht. Konturen lassen sich einfach definieren als eine Kurve entlang von Punkten, die alle dieselbe Farbe oder Intensität haben. Eine Folge von Konturen kann dann noch vereinfacht werden, wenn sie bspw. eine Linie bilden. Dann werden nur die beiden Endpunkte benötigt und der Rest verworfen [27]. Da das Schachbrett einen signifikanten Teil

des Bildes einnimmt, wird für die Anwendung in dieser Arbeit die zusammenhängende Folge an Konturpunkten mit der größten einschließenden Fläche gesucht.

Curve Approximation

Auf Basis der Folge an Konturpunkten des Schachbretts sollen jetzt die Eckpunkte des Bretts gefunden werden. Hierfür wird Curve Approximation genutzt. Die Folge wird also als (geschlossene) Kurve interpretiert und soll soweit vereinfacht werden können, dass nur noch die Eckpunkte übrig bleiben, da das Brett viereckig (genauer sogar trapezförmig) ist. Für die Approximation wird `cv.approxPolyDP()` genutzt, welche auf Basis des "Douglas-Peucker-Algorithmus" eine Kurve mit übergebenem ϵ approximiert. Bei diesem Algorithmus werden jeweils zwei Punkte (a und b) der Kurve genommen (anfangs Start- und Endpunkt) und der von diesen am weitesten entfernte (Punkt c) betrachtet. Ist die Entfernung $d \leq \epsilon$, werden die Punkte dazwischen entfernt. Gibt es keine Punkte dazwischen, wird aufgehört. Ist $d > \epsilon$, wird die Folge nach dem Prinzip Teile-und-herrsche aufgeteilt nach a bis b und b bis c und dann rekursiv erneut durchgeführt [33] (veranschaulicht in Abb. 5.9).

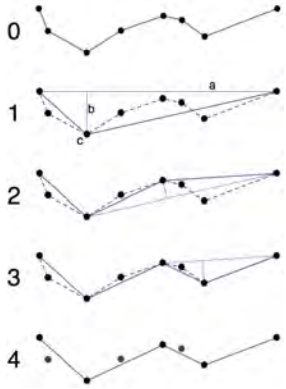
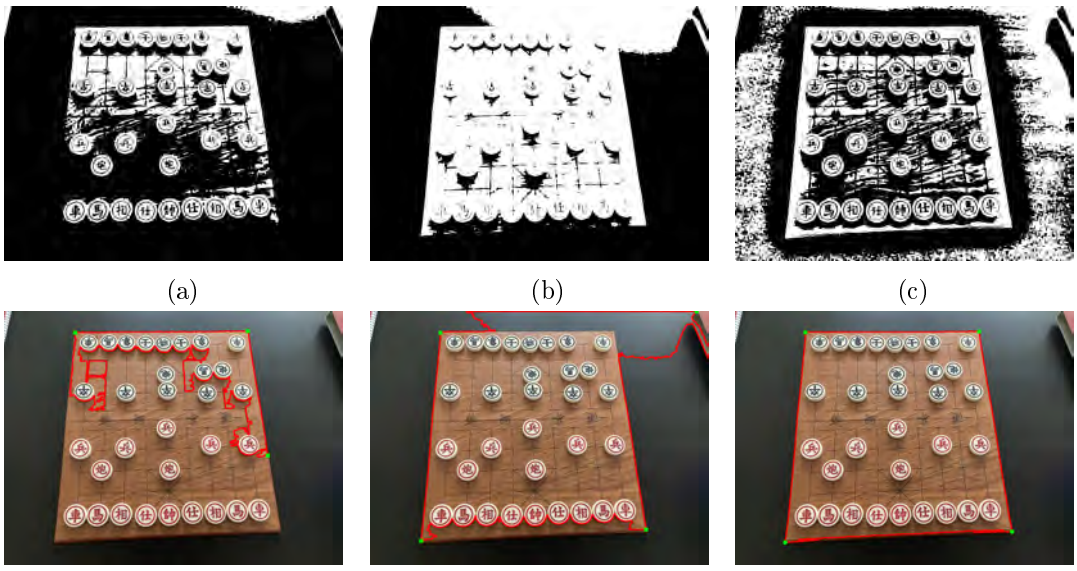


Abbildung 5.9: Veranschaulichung Douglas-Peucker-Algorithmus (Quelle: https://de.wikipedia.org/wiki/Datei:Douglas_Peucker.png, Zugriff Juli 2023)

Anwendung auf Bilder des Datensatzes

Um das Brett zu finden, wird nach Resizing, Grayscaling und Anwenden eines Gaussschen Filters adaptives Thresholding angewendet, die Konturen des Bretts gefunden und approximiert/geglättet, um die Eckpunkte zu ermitteln.

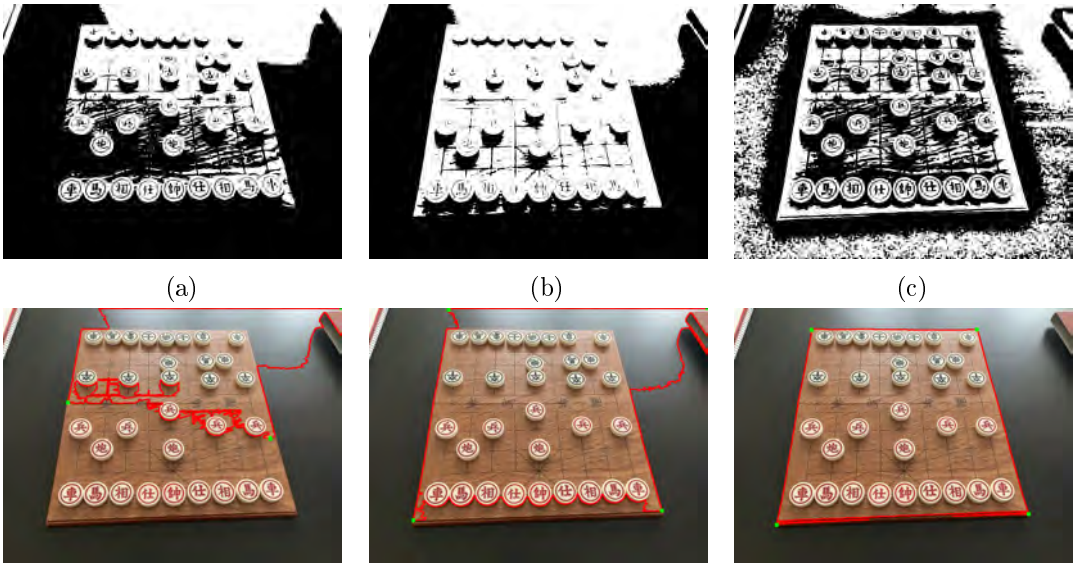
Beim Adaptive Thresholding wird das Gausssche Verfahren für die Threshold-Bestimmung der Nachbarschaft und das binäre Verfahren zum Setzen der Pixel genutzt bzw. das binär inverse, falls das Bild tendenziell heller ist (hier definiert durch Durchschnitt der Grauwerte größer als 132). Als *block size* (Größe der Nachbarschaft) haben große Werte die besten Ergebnisse erzielt, festgelegt wird dafür *block size*$= \frac{\sqrt{(width*height)}}{4} + 1$, also ein Viertel der Wurzel der Bildgröße (+ 1, weil die block size ungerade sein muss, um einen einzelnen Pixel im Mittelpunkt zu haben).
Für das Finden der Konturen wird `cv.findContours()` mit der obig erwähnten Vereinfachung der Linie (in OpenCV die Flag `cv.CHAIN_APPROX_SIMPLE`) genutzt und für verschachtelt auftretene Konturen wird eine Hierarchie aufgebaut (mit `cv.RETR_TREE`).
Zuletzt erfolgt die Approximation mit einem ϵ von 5% des Durchmessers der Konturenform.
Durchgeführt auf zwei Bildern aus unterschiedlichen Winkeln mit den anderen Thresholding-Methoden (global binär und Otsu) als Vergleich dargestellt in Abb. 5.10 und 5.11.



(a) (b) (c)

(d) (e) (f)

Abbildung 5.10: Bsp. 1: Anwendung Thresholding, Contour Finding (rot) und Eckpunkte durch Approximation (grün) (a, d) binär (b, e) Otsu (c, f) adaptive



(a) (b) (c)

(d) (e) (f)

Abbildung 5.11: Bsp. 2: Anwendung Thresholding, Contour Finding (rot) und Eckpunkte durch Approximation (grün) (a, d) binär (b, e) Otsu (c, f) adaptive

5.1.4 Entzerrung

Das gefundene Brett soll im letzten Schritt dieses Abschnitts entzerrt werden, das heißt, Verzerrungen sollen so eliminiert werden, dass das Brett senkrecht zur Bildebene steht. Anschließend wird das Bild so zugeschnitten, dass nur noch das Brett zu sehen ist, der Rest des Bildes wird nicht benötigt. Genauer wird die *Homographie* gesucht, eine 3x3 Matrix, welche Punkte einer projektiven Ebene (Bildfläche) auf die einer anderen abbildet [12].

Mathematisch ausgedrückt für die Beziehung der Punkte (x, y) und (x', y') (aus [24]):

$$s \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ mit } H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \text{ und } s \neq 0 \text{ (Konstante/Skalierungsfaktor).}$$

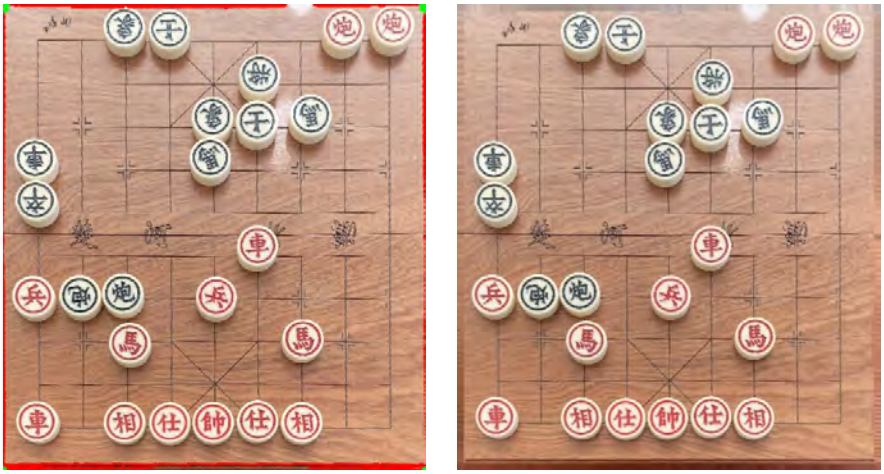
Da kein zweites Bild für die Abbildung existiert, werden imaginäre Punkte eines zweiten Bildes definiert, welches genau dessen Eckpunkte sind. Es werden also die Eckpunkte des Bretts auf die Eckpunkte eines neuen Bildes abgebildet, sodass dieses resultierend nur noch das Brett beinhaltet. Da die Punkte der beiden projektiven Ebenen bekannt bzw. definiert sind, muss die Homographie bestimmt werden. In OpenCV ist dies möglich mit der Methode `cv.findHomography()`. Damit wird die Homographie oder auch perspektivische

Transformation H berechnet mit:

$$s_i \begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} \sim H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix},$$

wobei der *back projection error* $\sum_i (x_i' - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}}) + (y_i' - \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}})$ minimiert wird[1]. Weil eine Abbildung von nur vier Punkte-Paaren benötigt wird, reicht hier auch die Methode `cv.getPerspectiveTransform()` aus, die konkret für diesen Fall definiert ist. Mittels `cv.warpPerspective()` wird dann die tatsächliche Entzerrung mit der ermittelten Homographie-Matrix durchgeführt.

Als Größe des neuen Bildes und damit auch der Festlegung der neuen Eckpunkte, wird 550x500 Pixel bestimmt. Der Grund dafür liegt darin, dass das Brett 9x8 Reihen x Spalten an Quadraten plus Rändern an den Seiten (die wie Quadrate behandelt werden) besitzt, damit also 11x10 Reihen x Spalten. Damit im nächsten Abschnitt des Algorithmus (5.2) die Ausschnitte gleich groß sein können, soll jedes Quadrat des Bretts 50x50 Pixel groß sein. Aufgrund des Verhältnisses von Rand zu Quadrat auf dem realen Brett (3cm:3,5cm), wird das Bild um einen künstlichen Rand erweitert (mittels `cv.copyMakeBorder()`) und dann wieder auf 550x500 reduziert, um tatsächlich gleich große Ausschnitte machen zu können (Veranschaulichung in Abb. 5.12).



(a) (b)

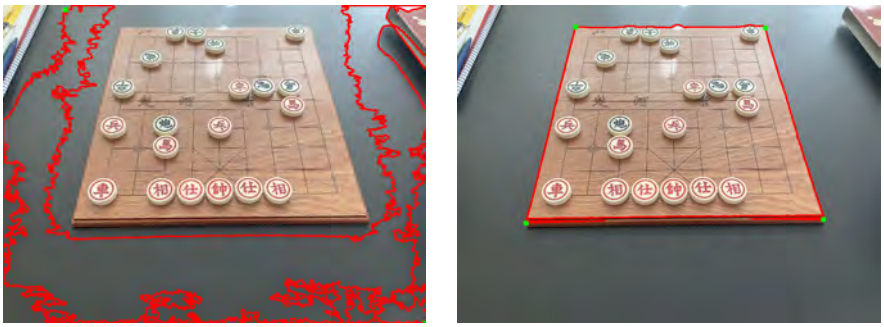
Abbildung 5.12: Resultierendes Bild des entzerrten Bretts (a) mit Konturen und Eckpunkten (b) mit künstlichem Rand

[1] aus https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html, Zugriff Juli 2023

5.1.5 Fehlerbehandlung

Während mit dieser Methode viele Bretter erkannt werden, kommt es bei anderen noch zu Misserfolgen. Hauptsächlich liegt dies an ungünstigen Lichtverhältnissen oder einem flachen Aufnahmewinkel. Um Fehler zu reduzieren und um nicht mit Bildern weiter zu arbeiten, bei denen fälschlicherweise das Brett als erkannt angenommen wird, wurden für die zwei Fehlerklassen passende Behandlungen eingebaut:

1. **Falsche Konturen bzw. Anzahl Ecken ungleich vier.** Ergibt die Approximation nicht vier Eckpunkte, kann davon ausgegangen werden, dass die Konturen des Bretts nicht korrekt gefunden wurden. In dem Fall wird das Adaptive Thresholding erneut ausgeführt, jedoch mit doppelter block size. Ergibt die Approximation anschließend weiterhin nicht vier Eckpunkte, wird abgebrochen und über den Fehler informiert. Das Bild soll dann unter besseren Rahmenbedingungen (siehe 6.3) erneut aufgenommen werden. Mit dieser Behandlung werden einige vorher verworfene Bretter doch erkannt (Beispiel Abb. 5.13).



(a) (b)

Abbildung 5.13: Brett mit (a) falschen Konturen (b) Konturen nach Fehlerbehandlung

2. **Vier Ecken, aber nicht die des Bretts.** Sind vier Ecken gefunden worden, kann nicht direkt davon ausgegangen werden, dass diese auch die Eckpunkte des Bretts sind. Um das zu überprüfen, werden perspektivische Eigenschaften betrachtet. Ist die Perspektive einer Aufnahme vom Brett nicht top-down, bildet das Brett im Bild eine Trapezform. Es werden also jeweils für die beiden oberen und unteren Punkte berechnet, ob die Linie zwischen ihnen näherungsweise dieselbe Steigung besitzt (für ein $\epsilon = 0.05$ absolut) und ob die vertikale bzw. horizontale Differenz der Distanzen zwischen den oberen Punkten nicht größer ist als die zwischen den unteren Punkten. Es ist nämlich möglich, dass die Linien annähernd parallel sind,

aber die x- oder y-Werte der Punkte der oberen Linie weiter auseinander liegen als die der unteren. Ist eine dieser Eigenschaften nicht erfüllt, sind die Punkte nicht die gewünschten gewesen und es wird wie in Punkt 1 weiter fortgefahren.
Da durch diese Überprüfung häufig Bretter aus der top-down-Perspektive als nicht trapezförmig identifiziert werden, wird noch geprüft, dass die Abstände der Konturen zu den Rändern oben und unten bzw. rechts und links sich nicht zu stark unterscheiden. Bei Brettern mit "schlechten" Konturen kommt es häufig vor, dass die Konturen bis zum Bildrand reichen, weil die Lichtverhältnisse den Brettrand schwierig vom Hintergrund unterscheidbar machen (Beispiele in Abb. 5.14).

(a) (b) (c)

Abbildung 5.14: Vier Ecken, aber (a) obere Steigung höher (b) ähnliche Steigung, aber horizontale Differenz höher (c) ähnliche Steigung, aber vertikale Differenz höher

5.2 Bildausschnitte

Wurde im vorherigen Schritt das Brett korrekt erkannt und entzerrt, erfolgt im nächsten Schritt das Ausschneiden der einzelnen Figuren in einer einheitlichen Größe, um diese dann im nächsten Kapitel 5.3 einzeln klassifizieren zu können.
Es wurde bereits sichergestellt, dass alle Quadrate inklusive des Rands dieselbe Höhe und Breite von 50 Pixeln besitzen. Die Figuren stehen jedoch nicht auf den Quadraten selbst, sondern auf ihren Schnittpunkten, konkret dem Schnittpunkt von vier umliegenden Quadraten. Es muss also ein größerer Ausschnitt gewählt werden, als die Quadrate selbst, um die Figur erkennen zu können. Im Rahmen dieser Arbeit wurde festgelegt, jeweils genau das 2x2-Raster auszuschneiden, in dem sich potenziell eine Figur im Mittelpunkt befindet. Die Ausschnitte sind demnach 100x100 Pixel und zeigen möglicherweise auch Teile von angrenzenden Figuren. Für den Ausschnitt ist aber nur die Figur im Mittel-

punkt interessant, also wird noch ein Rand der Breite 15 Pixel abgeschnitten, um den Fokus auf den Mittelpunkt zu setzen.

Mit diesem Vorgehen lassen sich alle Felder (also Schnittpunkte) des Spielfelds einzeln extrahieren. Es ergeben sich 9x10 = 90 Bilder á 70x70px Größe. Wichtig ist hierbei auch, dass die Ausschnitte des Bretts von links nach rechts und oben nach unten (also in natürlicher Leserichtung) durchgeführt werden, da sich in dieser Reihenfolge auch die Stellung im Kapitel 5.4 zusammenbaut. Beispiele für ausgeschnittene Figuren lassen sich in Abb. 5.15 sehen.

(a) (b) (c)

Abbildung 5.15: Ausschnitte Figuren (a) in der Ecke (b) leeres Feld (c) Figur mit Nachbarn

5.3 Figurenklassifizierung

Die extrahierten Figuren sollen nun klassifiziert werden, um die Stellung bestimmen zu können. Dafür werden im Vorfeld Trainingsdaten benötigt, um ein passendes neuronales Netz zu trainieren. Dabei ist das Ziel, eine hohe Genauigkeit der Vorhersagen erreichen zu können.

5.3.1 Trainingsdaten

Wie schon bei der Suche nach Veröffentlichungen zu diesem Thema, gibt es auch hier wenig Erfolg bei der Suche eines Datensatzes für XiangQi-Figuren. Das stellt jedoch kein großes Problem dar, weil auch hier die Möglichkeit besteht, den eigenen Datensatz zu nutzen, um einen neuen passenden zu generieren. Der Algorithmus bis zu diesem Schritt wird auf alle korrekt erkannten Bretter angewendet und die Figurenausschnitte gespeichert.

Mit zu dem Zeitpunkt 150 erkannten Brettern und 90 Ausschnitten pro Brett ergeben sich 13.500 Bilder der Felder. Da die Stellung regelmäßig variiert wurde, kommen einige Figuren etwas häufiger vor als andere (da diese schon geschlagen wurden). Generell gibt es zwei Vorkommen jeder Figur auf jeder Seite, außer der Generäle (jeweils einen) und der Soldaten (jeweils fünf). Die resultierenden Bilder wurden dann händisch analysiert. Wenn eine Figur eindeutig zu identifizieren war, wurde sie mit dem passenden Label (Klassenname) versehen. Daraus ergaben sich 14 Klassen (sieben Figuren auf jeder Seite) mit jeweils ca. 100 bis 340 Bildern plus zusätzlich die Klasse für ein leeres Feld (genannt "Empty"). Von der Klasse Empty gibt es offensichtlich erheblich mehr Bilder (>10.000), da selbst bei der Startaufstellung 58 von 90 Feldern leer sind. Um den Datensatz ausbalanciert zu halten, wurde hier noch einmal aussortiert, sodass auf letztendlich ca. 450 möglichst diverse Bilder dieser Klasse zugegriffen werden konnte. "Divers" bedeutet hierbei, dass alle möglichen Bereiche des Bretts und verschiedene Kombinationen benachbarter Figuren vorkommen. Mit den insgesamt knapp 3000 Bildern an Trainingsdaten konnte dann das *Supervised Learning* (überwachtes Lernen) in Form von Klassifizierung gestartet werden. Im nächsten Abschnitt werden CNNs allgemein und dann eigene konkrete Modelle und Ansätze vorgestellt.

5.3.2 Klassifizierung mit CNN

Besonders gut für Mustererkennung und damit auch für Klassifizierung von Bildern eignen sich *Convolutional Neural Networks* (CNNs). Der grundlegende Aufbau eines CNNs besteht aus 5 *Layers* (Schichten). Die erste Schicht ist das Input Layer, wobei die Input-Dimensionen der Neuronen die Bilddimensionen darstellen, also Höhe x Breite x Tiefe, mit Tiefe definiert als Anzahl der Kanäle des Bildes. Anschließend folgt ein *Convolutional Layer*, in dem der Output der Neuronen durch Berechnungen einzelner Regionen des Bildes mit zugehörigen *weights* (Gewichten) bestimmt wird. Als Aktivierungsfunktionen werden *rectified linear units (ReLus)* genutzt. Als nächstes folgt das *Pooling Layer*, in dem eine Dimensionsreduktion durchgeführt wird, das Bild entsprechend durch Abbildung einer Pixelregion auf einen einzelnen herunter gerechnet wird. An letzter Stelle befinden sich *fully connected layers* (auch *Dense* genannt), wie sie in "klassischen" neuronalen Netzen vorkommen, welche dann die Werte für die n Klassen für die Klassifikation berechnen (aus [31], graphische Darstellung Abb. 5.16).

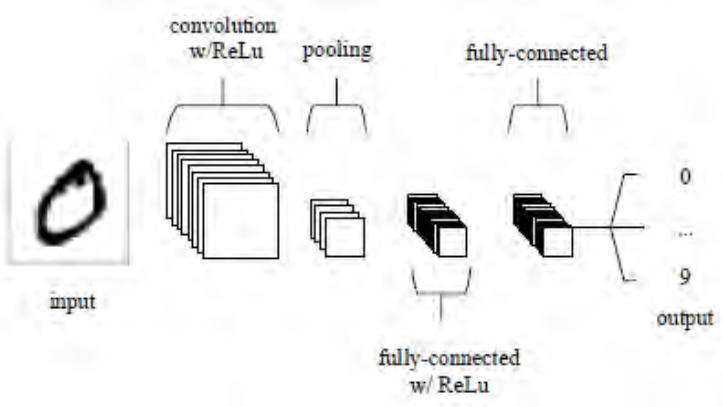


Abbildung 5.16: Architektur eines einfachen CNN (entnommen aus [31])

5.3.3 Erstes Modell

Für alle Modelle sind die Input-Dimensionen 70x70x3 und die Output-Dimensionen bzw. Klassen 1x1x15 (je sieben Figuren pro Seite und die Klasse Empty). Als Bibliothek für Machine Learning wurde Tensorflow mit Keras[2] in der Version 2.14.0 genutzt. Alle Trainings wurden über Google Colab[3] durchgeführt, da dort leicht zugänglich Rechenleistung zur Verfügung gestellt wird, um den Trainingsprozess zu vereinfachen und zu beschleunigen.

Das erste Modell besteht aus zwei aufeinanderfolgenden Convolutional + MaxPooling Layern für die ersten Berechnungen und Dimensionsreduktionen des Bildes, sowie einem *Flatten Layer* für die Dimensionsreduktion der Neuronen von 3D auf 1D, um dann mit zwei Dense Layers, zum einen mit ReLu-Activation, zum anderen mit *Softmax* für den Output, verbunden zu werden. Zwischen den beiden Dense Layers befindet sich außerdem ein *Dropout* Layer, welches zufällig Neuronen ausschaltet, um die Gefahr von *Overfitting* (Spezialisierung auf die Trainingsdaten statt Generalisierung für neue Daten) zu reduzieren. Als Optimizer wurde *Adam*, als *Loss function* wurde *categorical crossentropy* gewählt. Dieses Modell wurde 30 Epochen lang mit Validierungsdaten zur Überwachung trainiert, kam aber nicht über eine *Accuracy* (Genauigkeit der Vorhersagen) von 38% hinaus. Abgewandelte Konfigurationen dieses Modells erreichten auch keine höheren Werte, also wurde ein anderer Ansatz benötigt.

[2] https://www.tensorflow.org/, Zugriff Juli 2023
[3] https://colab.research.google.com/, Zugriff Juli 2023

5.3.4 Modell auf ImageNet-Basis

Ein Ansatz zur Verbesserung bestand darin, ein vortrainiertes Netzwerk zu nutzen, wobei die Wahl auf ein *ImageNet*-Modell fiel.
ImageNet ist eine Idee von Fei-Fei Li, einer heutigen Professorin für Computer Science an der Stanford University. Die grundlegende Idee ist, nicht die Algorithmen, sondern den Datensatz zu verbessern. Aus ihrer Sicht reflektierten die Datensätze die Welt nicht gut genug, also entwickelte sie mit ihrem Team einen Datensatz von Bildern tausender verschiedener Objekte und Kategorien. Auf Basis dieses Datensatzes fand dann von 2010 bis 2017 die *ImageNet Large Scale Visual Recognition Challenge (ILSVRC)* statt, in der Algorithmen zur Objekterkennung und Bildklassifizierung auf Basis dessen bewertet wurden [13][14][15]. Eins dieser Modelle, auf das die Wahl in dieser Arbeit gefallen ist (unter anderem, weil es einfach in Tensorflow Keras zu importieren ist), heißt *VGG19*, eine verbesserte Version von *VGG16*, welches in der ILSVRC 2014 den zweiten Platz belegte [34]. An die Basis dieses Modells konnte das im Rahmen dieser Arbeit eigen entwickelte Modell angehängt werden, bestehend aus einem Flatten Layer und zwei Dense Layers (Aktivierung ReLu und Softmax) mit einem Dropout Layer dazwischen.
Mit dieser recht simplen Konfiguration konnte mit 30 Epochen Training schon eine enorme Verbesserung erreicht werden mit einer Accuracy von knapp 98%. Da bei 90 Vorhersagen (9x10 Felder) mit 98% Genauigkeit trotzdem mit etwa 2 Fehlern gerechnet werden muss, sollten noch weitere Verbesserungen gefunden werden, um die Fehler weiter zu reduzieren.

5.3.5 Focal Loss

Ein weiterer Ansatz zur Verbesserung bestand in der Verwendung von *focal Loss*. Focal Loss ist eine Loss function, die dabei hilft, Probleme der Klassenungleichgewichte zu lösen, bei denen bestimmte Klassen im Datensatz häufiger vorkommen als andere [21].
Dieser Ansatz bewirkte bereits eine minimale, aber signifikante Verbesserung und konnte 99% Accuracy erreichen.

5.3.6 Hierarchical Loss

Ein letzter Ansatz, auf Basis eines Vorschlags von Prof. Dr. Peer Stelldinger, war die Nutzung von *hierarchical Loss*. Die Idee hierbei ist, das Problem der Klassifizierung in mehrere Hierarchieebenen aufzuteilen. In dem Beispiel der Autoren ist bei der Erkennung eines Gesichts die erste Hierarchiebene "background" und "head", die zweite besteht aus "hair" und "face" (abgeleitet von "head") [23]. Ein Beispiel zu sehen in Abb. 5.17.

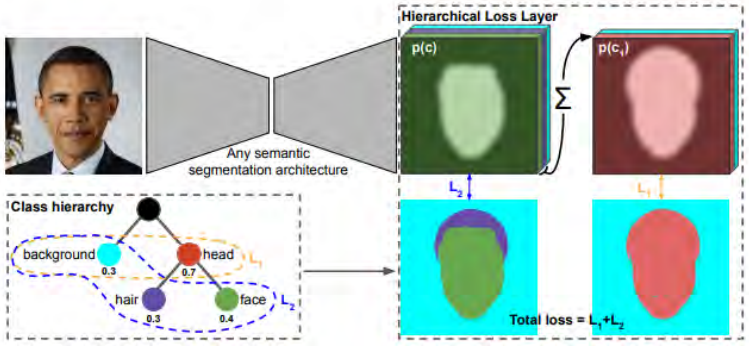


Abbildung 5.17: Grundlegende Idee von hierarchical loss (entnommen aus [23])

Für den Anwendungsfall dieser Arbeit wurden folgende Hierarchien getestet.
1. Ebene "has-piece" ("Ist eine Figur auf diesem Feld?"), 2. Ebene "red-or-black" ("Ist die Figur rot oder schwarz?") und 3. Ebene "piece-class" ("Was ist die genaue Klasse?"). Mit verschiedenen Kombinationen der Ebenen wurden jeweils Trainings durchgeführt. Da leere Felder jedoch sowieso bereits mit sehr hoher Sicherheit erkannt wurden, konnte die erste Ebene ("has-piece") entfernt und mit den anderen Ebenen weitergearbeitet werden. Eine höhere Accuracy konnte hiermit jedoch leider nicht erreicht werden, also fiel die endgültige Wahl des Modells auf das im vorherigen Abschnitts beschriebenen mit VGG19 und Focal Loss. Ein Vergleich des Trainingsverlaufs der Modelle mit Accuracy über Epochen lässt sich in Abb. 5.18 sehen, eine *Confusion Matrix* zum besten Modell absolut in Abb. 5.19 und relativ in Abb. 5.20.

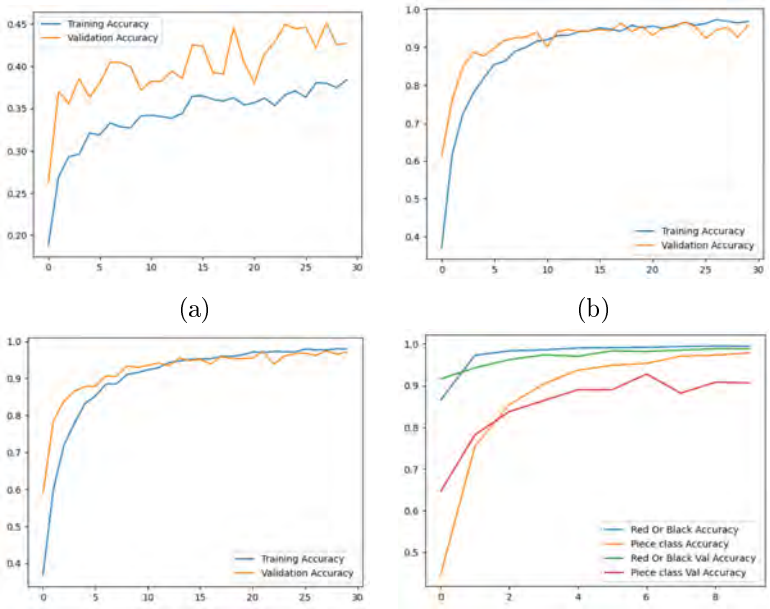


Abbildung 5.18: Vergleich der Modelle (a) erstes Modell (b) Modell auf VGG19 (c) Modell mit Focal Loss (d) Modell mit Red-Or-Black- und Piece-Class-Hierarchie

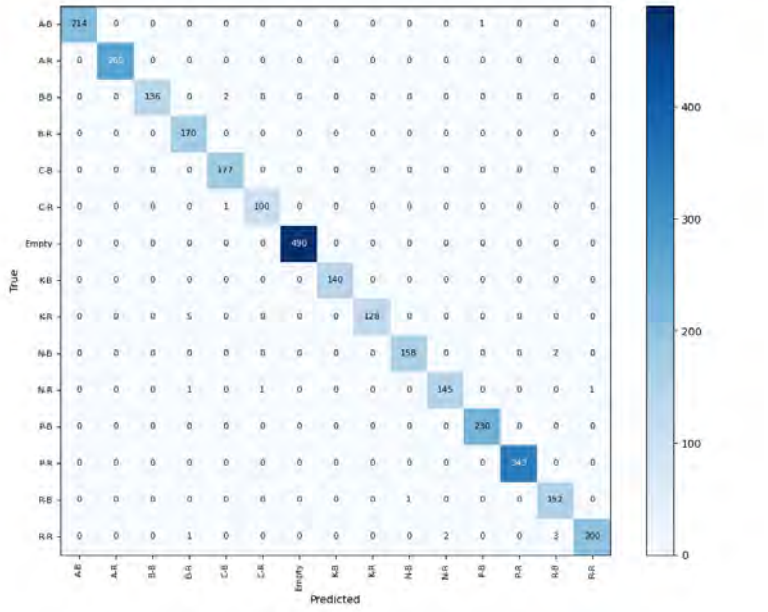


Abbildung 5.19: Confusion Matrix des besten Modells absolut

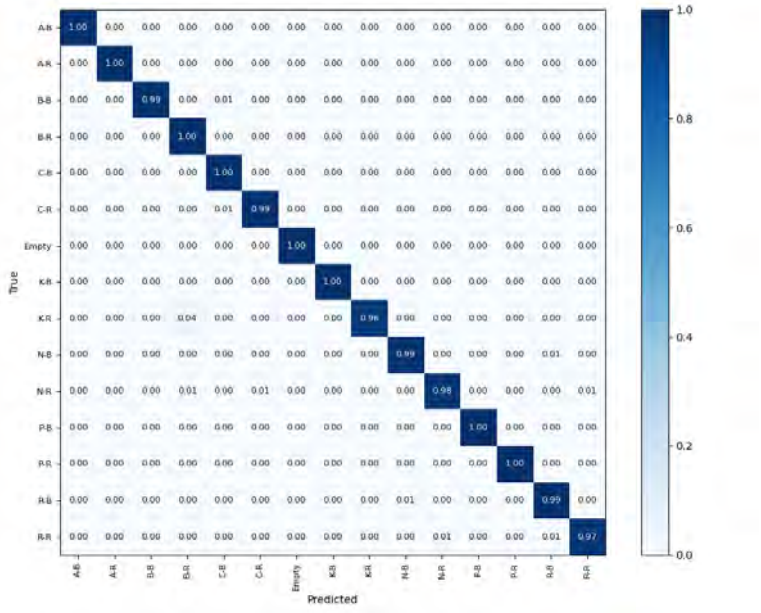


Abbildung 5.20: Confusion Matrix des besten Modells relativ

5.4 Stellung als FEN

Mithilfe des trainierten neuronalen Netzes, können nun alle Felder auf dem Brett klassifiziert werden. Um die vorliegende Stellung darzustellen und für eine Engine verständlich zu machen, wird für diese eine kompakte Repräsentation benötigt. Die heute etablierteste Notation ist die "Forsyth-Edwards Notation" (FEN), entwickelt von Steven J. Edwards, der diese auf der Basis einer älteren Notation von David Forsyth aufgebaut hat [6].

5.4.1 FEN allgemein

Die FEN wird dargestellt als ein String von ASCII-Zeichen. Formal definiert sieht ein FEN-String so aus:

`<FEN> ::= <Piece Placement>' ' <Side to move>' ' <Castling ability>' ' <En passant target square>' ' <Halfmove clock>' ' <Fullmove counter>.`

Dabei steht die konkrete Position der Figuren in `<Piece Placement>` und zusätzlich gibt es noch Informationen über den Spielstand, ob beispielsweise die Rochade für Spieler möglich ist (in `<Castling ability>`). Interessant ist aber die Definition der Stellung selbst.

Formal ist sie so beschrieben:

```
<Piece Placement> ::=
          <rank8>'/'<rank7>'/'<rank6>'/'<rank5>'/'<rank4>'/'<rank3>'/'<rank2>'/'<rank1>
<ranki> ::= [<digit17>]<piece> [<digit17>]<piece> [<digit17>] | '8'
<piece> ::= <white Piece> | <black Piece>
<digit17> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7'
<white Piece> ::= 'P' | 'N' | 'B' | 'R' | 'Q' | 'K'
<black Piece> ::= 'p' | 'n' | 'b' | 'r' | 'q' | 'k'
```

Dabei wird das Brett aus der Sicht von weiß vom letzten Rang (Reihe) von links nach rechts und oben nach unten durchgegangen und die Figuren jedes Felds notiert. Jede Figur hat dabei eine Abkürzung von einem Zeichen, wobei weiße Figuren groß- und schwarze Figuren kleingeschrieben sind. Die Abkürzung ergibt sich durch den Anfangsbuchstaben des englischen Wortes der Figur, also *(P)Pawn, (B)Bishop, (R)Rook, (Q)Queen, (K)King* und *(N)Knight* (da das "K" schon vergeben ist). Felder ohne Figuren werden innerhalb einer Reihe hochgezählt, bis wieder eine Figur kommt oder der Rand erreicht wird. Ränge werden durch "/" unterteilt. Die Startaufstellung im Schach ergibt sich damit also als `rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1` (Abschnitt aus [5]).

5.4.2 FEN für XiangQi

XiangQi hat zwar etwas andere Regeln und Figuren, jedoch kann hier auch die FEN angewendet werden. Laut den offiziellen Regeln der *World Xiangqi Federation (WXF)*, sind die englischen Abkürzungen für die Figuren wie folgt: *C(Cannon), H(Horse), R(Chariot), E(Elefant), A(Advisor), P(Pawn/Soldier), K(King/General)* (aus [36], Kapitel 2.7.4, Seite 16). Die in dieser Arbeit genutzte Engine (siehe 5.5) nutzt jedoch die westliche Schachnotation, basierend auf den Ähnlichkeiten der Figuren (siehe 3.2), das Pferd ist also "N" statt "H" und der Elefant "B" statt "E". Die Startaufstellung für XiangQi sieht demnach wie folgt aus: `rnbakabnr/9/1c5c1/p1p1p1p1p/9/9/P1P1P1P1P/1C5C1/9/RNBAKABNR w`.
Das "w" am Ende steht für Weiß (bzw. Rot in XiangQi) am Zug. Weitere Informationen der FEN sind für die Engine nicht weiter relevant, da die enthaltenen Regeln im chinesischen Schach nicht existieren.

5.4.3 Anwendung FEN

Den FEN-String in diesem Algorithmus zusammenzubauen ist sehr gradlinig. Da bereits sichergestellt wurde, dass die Bilder der Figuren in der richtigen Reihenfolge ausgeschnitten werden (siehe 5.2), kann das neuronale Netz die Klassifizierung durchführen und der String direkt gebaut werden. Bei Klassifizierung eines leeren Feldes wird ein Zähler hochgezählt, bis wieder eine Figur oder das Ende der Reihe kommt. Zum Schluss müssen noch folgende Eigenschaften sichergestellt werden:
Zum einen muss wie im FEN des westlichen Schachs die schwarze Seite immer zuerst im String vorkommen, weil sie "oben" aus der Sicht von weiß/rot ist. Da sich die Generäle/Könige nur innerhalb ihrer Paläste befinden können, wird ihre Position überprüft und der String umgedreht, falls nötig.
Zum anderen muss bestimmt werden, welche Seite am Zug ist, also welche*r Spieler*in einen Zugvorschlag erhalten möchte. Es gibt keine Informationen über den bisherigen Verlauf des Spiels, das ist jedoch auch nicht notwendig. Die Grundannahme ist, dass das Bild aus der eigenen Spielperspektive aufgenommen wird, demnach ist die Seite am Zug, die sich auf der unteren Seite des Bretts befindet. Auch das lässt sich durch die Position der Generäle bestimmen.
Damit ist der FEN-String fertig, es könnte jedoch noch sein, dass er fehlerhaft ist, aufgrund falscher Klassifizierung.

5.4.4 Fehlerbehandlung

Auch bei einer Genauigkeit von 99% des neuronalen Netzes muss bei 90 Feldern mit mind. einem Fehler gerechnet werden. Den konkreten Fehler ausfindig zu machen und zu korrigieren, ist für das Programm unmöglich. Für ein gegebenes Bild einer Figur wird die Klassifizierung immer dieselbe sein. Jedoch kann der FEN-String auf Gültigkeit überprüft werden. Dabei gibt es zwei mögliche Klassen von Fehlern, die auftreten können:

1. **Anzahl Vorkommen einer Figur**. Jede Figur gibt es nur bestimmt oft. Ist ein gültiges Vorkommen über- oder unterschritten, kann von einem fehlerhaften FEN-String und damit einer fehlerhaften Klassifizierung ausgegangen werden. Konkret bedeutet das, falls es mehr als fünf Soldaten, nicht genau einen König oder mehr als zwei mal eine andere Figur einer Seite gibt, ist die ermittelte Stellung ungültig.

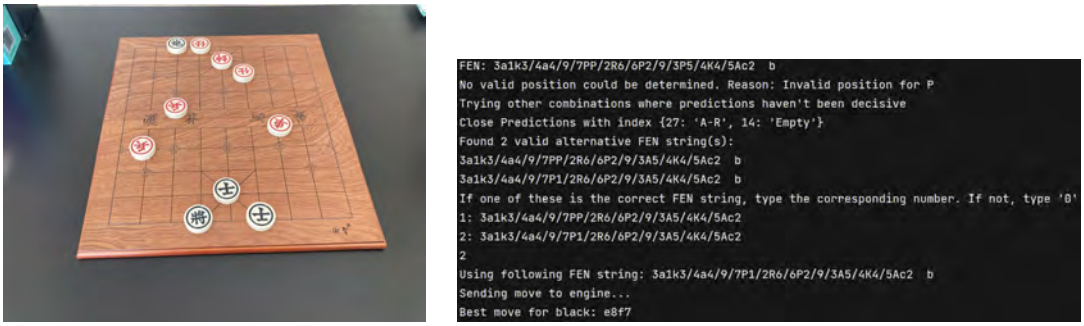
2. **Ungültige Positionierung einer Figur**. Wie bereits in Kapitel 3 erläutert, gibt es für einige Figuren Einschränkungen ihrer Position auf dem Feld. Diese Einschränkungen sind gegeben durch den Fluss, den Palast und die Startaufstellung. Ein General und seine Berater können also nicht außerhalb des Palastes sein, die Elefanten können den Fluss nicht überqueren und sich nur auf durchgezählt geraden Reihen befinden (da sie immer zwei Schritte diagonal machen) und Soldaten können nicht rückwärts laufen, besitzen also eine "Mindestreihe", auf der sie sich befinden können. Gibt es hier ungültige Positionierungen, ist der FEN-String ungültig.

Es muss jedoch nicht direkt abgebrochen werden, falls ein falscher FEN-String entstanden ist. Um möglicherweise doch noch die korrekte Stellung bestimmen zu können, wird folgendes gemacht:

1. Bei jeder Klassifizierung wird geschaut, ob sie eindeutig gewesen ist. "Eindeutig" wird dadurch definiert, dass das Verhältnis der Wahrscheinlichkeiten von der zweithöchsten zur höchsten Prognose einen gewissen Wert überschreitet. Das Verhältnis wird festgelegt als 2:5 bzw. 40%. Das bedeutet, wenn bspw. die beiden höchsten Wahrscheinlichkeiten 70% und 30% betragen, ist das Verhältnis $\frac{30}{70} = \frac{3}{7} > 0,4$ und damit eine uneindeutige Klassifizierung. Gibt es also eine uneindeutige Klassifizierung, wird die erste zum FEN-String hinzugefügt und die zweite mit aktuellem Index im String gespeichert.

2. Ist der FEN-String am Ende nun ungültig, werden alle Kombinationen des FEN-Strings mit den gespeicherten alternativen Figuren ausprobiert und auf Gültigkeit geprüft. Ist kein gültiger String dabei, wird abgebrochen und der/die Nutzer*in informiert. Ist genau ein valider dabei, wird davon ausgegangen, dass es der passende ist, und weiter fortgefahren. Falls mehrere gültige gefunden werden, wird der/die Nutzer*in gefragt, ob und wenn ja, welcher der richtige ist und mit diesem fortgefahren. Falls der korrekte nicht dabei ist, wird abgebrochen und informiert (Beispielinteraktion in Abb. 5.21).

Der FEN-String kann valide, aber trotzdem nicht korrekt sein. Das ist gerade dann wahrscheinlich, wenn sich weniger Figuren auf dem Brett befinden. In solch einem Fall muss manuell überprüft werden, ob die Stellung korrekt erkannt wurde und im Zweifel eine erneute Aufnahme gemacht werden.



(a) (b)

Abbildung 5.21: Fehlerbehandlung bei mehreren FEN-Strings. (a) Spielsituation (b) Konsoleninteraktion

5.5 Zugvorschlag mit Engine

Im letzten Schritt des Algorithmus soll nun die ermittelte Stellung in FEN an eine Schachengine für chinesisches Schach gegeben werden. Dazu vorher eine Einleitung zu Schachengines im westlichen Schach und die Auswahl der Engine für diese Arbeit.

5.5.1 Schachengines und Auswahl

Mit der stetigen Weiterentwicklung von Computern, Rechenleistung, Speicherkapazität und Algorithmen entwickelten sich viele Schachcomputer, von denen es mittlerweile hunderte gibt, die Schachgroßmeister (der höchste Rang im Schach) schlagen können. Heutzutage werden sie dafür genutzt, Stellungen ausführlich zu analysieren und für Menschen Ideen zu generieren. Die grundlegende Funktionsweise ist dabei zum einen die Bewertung der aktuellen Stellung auf Basis von Figurenwerten und je nach Engine anderen Bewertungssystemen. Zum anderen gibt es die Suchphase, wobei sich die Zustände nach verschiedenen Zügen angeschaut und bewertet werden, um die besten Züge zu ermitteln [4].

Als weltweit stärkste Engine für westliches Schach hat sich *Stockfish*[4] etabliert. Bei den größten Wettbewerben für Schachengines, wie beispielsweise *TopChessEngineChampionship (TCEC)*[5] oder der *ComputerChessChampionship (CCC)*[6], konnte Stockfish in den

[4] https://stockfishchess.org/, Zugriff Juli 2023
[5] https://tcec-chess.com/, Zugriff Juli 2023
[6] https://www.chess.com/computer-chess-championship, Zugriff Juli 2023

letzten fünf Jahren fast immer den ersten Platz erreichen. Stockfish basiert auf *Alpha-beta pruning*, einem Suchalgorithmus für Zustandsbäume, und riesigen Datenbanken zu Eröffnungen und Endspielen.

Eine von Stockfish abgeleitete Engine für chinesisches Schach ist *Pikafish*[7], auf welche die Entscheidung in dieser Arbeit fiel, weil sie eine höhe Stärke besitzt, open-source verfügbar ist und eine einfache Kommunikationsschnittstelle bietet. Die Website ist auf Chinesisch, jedoch sind Informationen auf Englisch auf der GitHub-Projektseite verfügbar.

5.5.2 Kommunikation mit der Engine

Mit Pikafish werden unterschiedliche Engines zur Auswahl gestellt, je nach Rechenleistung auswählbar. Der Einfachheit halber wurde für diese Arbeit die grundlegende gewählt. Pikafish bietet keine graphische Oberfläche (GUI) an, die Kommunikation über die Konsole reicht jedoch vollkommen aus. Wie viele andere Engines auch, nutzt Pikafish das *Universal Chess Interface (UCI)* als Kommunikationsschnittstelle [7].
Darin sind Konsolen-Kommandos definiert, die ein*e Nutzer*in eingeben kann, und wie eine Engine damit umzugehen hat. Die wichtigsten Kommandos für diese Anwendung sind (Beispielinteraktion in Abb. 5.22):

- `position [ fen fenstring | startpos ]`. Hier kann die im vorherigen Schritt ermittelte Stellung für die Engine gesetzt werden. `startpos` wird nicht benötigt, es wird immer die Option `fen fenstring` genutzt.

- `go [depth <x>]`. Hiermit wird der Engine gesagt, dass sie für die festgelegte Stellung den besten Zug mit einer übergebenen *depth* (Tiefe) für den Suchbaum suchen soll. Wird keine Tiefe übergeben, würde endlos gesucht werden, bis `stop` geschrieben wird. Solch ein Verhalten ist für diese Anwendung nicht brauchbar, es wird also immer eine Tiefe mitgegeben, standardmäßig auf 15 festgelegt. Als Resultat liefert die Engine eine Konsolenausgabe mit der letzten Zeile in der Form

  ```
  bestmove <move> ...
  ```

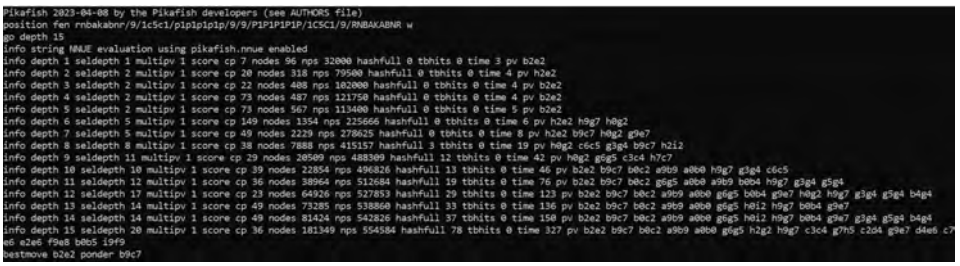
- `d`. Dieses Kommando ist nicht zwingend notwendig, hilft aber zur Veranschaulichung. Hierbei wird die aktuelle Stellung gut formatiert und lesbar mit Zeilen- und Kantenbeschriftung ausgegeben. Da die Notation für die Züge nicht in der verbreiteten *Standard Algebraic Notation* (SAN), sondern der *Pure algebraic coordinate*

[7] https://pikafish.org/, Zugriff Juli 2023

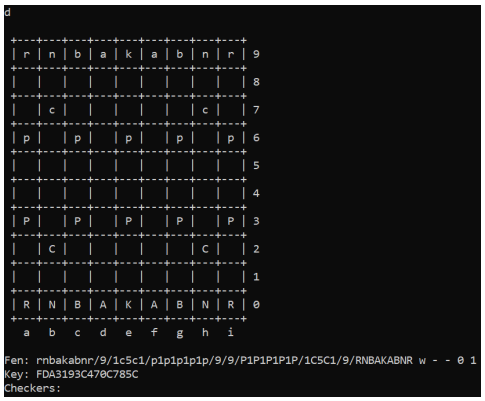
notation (genannt *long algebraic notation* in UCI) ausgegeben werden, kann diese Veranschaulichung zur Orientierung helfen. Bei dieser Notation werden, anders als in der SAN, nur die Koordinatenänderungen bezeichnet. Formal definiert:

```
<move descriptor> ::= <from square><to square>

<square> ::= <file letter><rank number>

<file letter> ::= 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'

<rank number> ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'
```

Dabei steht *file* für die Spalte und *rank* für die Zeile, jeweils aus Sicht von weiß [1]. Für XiangQi und damit auch für Pikafish gehen die Spalten von a bis i und die Zeilen von 0 bis 9 (aufgrund des größeren Bretts).



(a)



(b)

Abbildung 5.22: Konsoleninteraktion mit Pikafish. (a) Festlegung Stellung + bester Zug (b) Repräsentation der Stellung

Für die Kommunikation in dieser Anwendung wird Pikafish als Subprozess mit `stdin` und `stdout` als `PIPES` gestartet, die Kommandos mit ihren Parametern in `stdin` geschrieben und dann die Antwort inklusive des besten Zugs aus `stdout` gelesen.

Als weitere nutzerfreundliche Erweiterung wird der beste Zug außerdem in Form eines Pfeils auf das entzerrte Brett gemalt und dem*r Nutzer*in gezeigt, dass bei Unwissen über die Notation der Zug trotzdem verstanden werden kann.

6 Experimente

Wie gut funktioniert das Programm nun? Getestet wurde auf Brett- und Stellungserkennung, auf Bildern des gesamten Datensatzes und auf zusätzlichen Testbildern. Die Ergebnisse, Fehlerquellen und daraus resultierenden Rahmenbedingungen zur Benutzung werden in diesem Kapitel aufgezeigt.

6.1 Ergebnisse

Eigene Bilder und Testbilder sind hier noch einmal unterteilt, weil sie unterschiedlich zu interpretieren sind.

Ergebnisse auf Bildern des Datensatzes

Es wurden alle Bilder an das Programm gegeben und geprüft ob das Brett erkannt wird, ob nicht erkannte Bretter nach Fehlerbehandlung (in Tabelle *Handling*) erkannt werden (siehe 5.1.5), die Stellung korrekt bestimmt wird und bei falschen Stellungen nach Fehlerbehandlung die korrekte gefunden wird (siehe 5.4.4). Folgendes Ergebnis resultierte:

	Brett		**Stellung**	
	erkannt?	Handling erfolgreich?	erkannt?	Handling erfolgreich?
Num ok	151	11	89	10
Num no	41	30	73	63
Erfolgsrate	78.65%	26.83%	54.94%	13.70%
Gesamt	Bretterkennung 162/192 = 84.38%		Stellungserkennung 99/162 = 61.11%	

Es wird deutlich, dass die Bretterkennung überwiegend verlässlich funktioniert. Beim Erkennen der Stellung gibt es vereinzelt Misklassifikationen, teilweise auch ganze Reihen, größtenteils wird sie jedoch korrekt erkannt. Genaueres dazu in den Fehlerquellen (Abschnitt 6.2).

Ergebnisse auf Testbildern

Als Testbilder wurden solche ausgewählt, die einen anderen Hintergrund haben, die ein anderes Brett zeigen und welche, die nicht aus Spielerperspektive aufgenommen wurden. Externe Bilder sind entsprechend gekennzeichnet. Einige Testbilder wurden außerdem so zugeschnitten, dass sie hauptsächtlich das Brett zeigen. Sind Bilder aus der "Zuschauerperspektive" aufgenommen, also von der Seite, wurden sie rotiert, sodass die Sichtweise besser zur Anwendung passt.

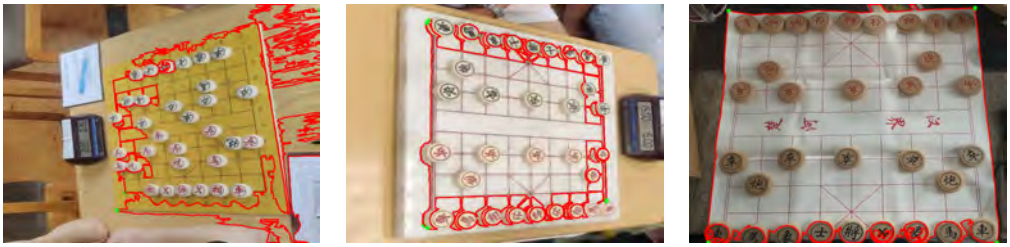
Abb. 6.1 zeigt Bilder und gefundene Konturen und Ecken des Bretts auf verschiedenen Oberflächen/ Hintergründen. Nur beim Glastisch stellten sich bei der Bretterkennung Schwierigkeiten ein. Auf den anderen beiden Bildern wurden die meisten, aber nicht alle Figuren korrekt erkannt.
Abb. 6.2 zeigt eine Bildauswahl mit gefundenen Konturen und Ecken. Die Bilder wurden extern generiert. Diese wurden anschließend zugeschnitten, um den Fokus auf das Brett und etwas günstigere Bedingungen zu haben. Bei diesen Bildern wird im Voraus davon ausgegangen, dass weder Brett- noch Stellungserkennung erfolgreich sein wird. Zum einen sind sie aus einem falschen Winkel aufgenommen, zum anderen besitzen sie anders gestaltete Figuren (mehr dazu in Abschnitt 6.3).

(a) (b) (c)

Abbildung 6.1: Bretterkennung eigenes Brett auf (a) weißer Tisch (b) Glastisch (c) Holztisch

6.2 Fehlerquellen

Die Gründe für entstandene Fehler bei beiden Arten von Erkennungen lassen sich sehr gut konkretisieren.



(a) (b) (c)

Abbildung 6.2: Bretterkennung fremdes Brett aus (a, b) seitlicher/ Zuschauerperspektive (c) auf Spielmatte

Bretterkennung

Wurde ein Brett nicht erkannt, lag es in 28 von 30 Fällen an der flachen Perspektive und/ oder den Lichtverhältnissen. Ein einziges, grundsätzlich korrekt erkanntes Brett fiel beim Test auf Trapezform durch und wurde deswegen fälschlicherweise verworfen. Allgemein wurden bei Bildern aus der Top-Down-Perspektive häufig keine Trapeze gefunden, wie in 5.1.5 bereits diskutiert. Die Testbilder haben gezeigt, dass bei einem gleichmäßigen Hintergrund eine Erkennung die beste Aussicht auf Erfolg hat. Außerdem können neben Gegenlicht auch Schatten die Erkennung stören.

Stellungserkennung

Bei falsch ermittelten Stellungen war die häufigste Ursache, dass aufgrund der Perspektive einige Figuren leere Felder verdeckt haben, die Klassifikation also eine Figur in zwei aufeinander folgenden Reihen erkennt, obwohl diese nur einmal auftaucht. Auch die erwähnten Lichtflecken machten Klassifikationen schwieriger. Die Experimente auf den Testbildern zeigten außerdem, dass Figuren eines anderen Spielsets schwieriger klassifiziert werden können, da das neuronale Netz nur auf Figuren des eigenen Bretts trainiert wurde.

6.3 Resultierende Rahmenbedingungen

Aus den Experimenten und ihren Ergebnissen und Fehlerquellen lassen sich folgende Rahmenbedingungen für die höchste Wahrscheinlichkeit auf Erfolg bei Benutzung dieses Programms festlegen:

1. Das Bild sollte aus der Perspektive eines Spielenden aufgenommen werden.
2. Der Winkel, aus dem das Bild aufgenommen wird, sollte nicht zu flach sein. Ein guter Anhaltspunkt ist, dass den oberen Rand des Bretts nicht die Figuren bilden, sondern der Rand des Bretts noch vollständig zu erkennen ist.
3. Es sollte möglichst einheitliche Lichtverhältnisse geben, das heißt kein Gegenlicht und möglichst auch wenig Schatten.
4. Ein großer Kontrast zwischen Brett und Untergrund, der außerdem möglichst gleichmäßig sein sollte, führt ebenfalls zu besseren Ergebnissen.
5. Für gute Ergebnisse sollte das Brett einen großen Teil des Bildes ausmachen.
6. (Am besten funktioniert der Algorithmus mit Figuren die dem Original im Datensatz möglichst ähnlich oder gleich sind, da das neuronale Netz nur meine Figuren kennt. Dazu im nächsten Kapitel zum Ausblick mehr.)

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Algorithmus vorgestellt, der anhand eines Bildes von einem Spielbrett aus dem chinesischen Schach das Brett und die vorliegende Stellung erkennt und mittels der Engine Pikafish den besten Zug vorschlägt.
Dabei wurde das Brett durch adaptives Thresholding, Finden von Konturen und Approximation dieser zum Finden der Eckpunkte, erkannt. Jedes einzelne Feld wurde einzeln und einheitlich ausgeschnitten, um dann die Figuren mit einem neuronalen Netz basierend auf ImageNet und mit Focal Loss zu klassifizieren. Mit den Klassifikationen wurde die Stellung in der Forsyth-Edwards-Notation zusammengebaut, sodass Pikafish die Stellung durch Konsoleneingabe verstehen und mit einer gegebenen Tiefe den besten Zug finden und ausgeben kann. Für den Fall, dass ein*e Nutzer*in die Notation nicht versteht, wird dieser Zug zusätzlich auf das Bild des Bretts aufgezeichnet. Unter Einhaltung einiger Rahmenbedingungen wird das gewünschte Verhalten mit hoher Sicherheit erreicht. Ein beispielhafter schrittweiser Ablauf des Programms befindet sich im Anhang A.

Wie die Experimente gezeigt haben, werden die besten Ergebnisse bei Bildern mit Brett und Figuren des in dieser Arbeit genutzten Spielsets erreicht. Andere Spielbretter können auch gut erkannt werden, jedoch ist es nicht unwahrscheinlich, dass andere Sets Figuren anderer Größe, Farben und Zeichenart der Schriftzeichen besitzen. Um auch flexibler gegenüber verschiedenen Brettern zu sein, können die Trainingsdaten durch Erweiterung um Figuren anderer Spielsets diversifiziert werden. Eine weitere mögliche Erweiterung wäre beim Ausschneiden der Felder möglich, indem sich hierbei mehr an die Perspektive angepasst wird. Eine letzte mögliche Anpassung liegt bei der Robustheit gegenüber verschiedenen Lichtverhältnissen. Möglicherweise ließe sich hier durch Analyse der Bilddaten dynamisch eine optimalere block size finden oder vielleicht sogar eine passende Kombination an Parametern für einen Ansatz mit Canny Edge und Hough Transform.

Literaturverzeichnis

[1] : *Algebraic Chess Notation*. – URL https://www.chessprogramming.org/Algebraic_Chess_Notation. – Zugriffsdatum: 2023-07

[2] : *"凯撒杯"首届世界青少年象棋锦标赛*. – URL https://www.yunbisai.com/tpl/eventFeatures/eventDetail-985.html#groupID=3471. – Zugriffsdatum: 2023-07

[3] : *Chinese Chess on Xianqi.com*. – URL https://www.xiangqi.com/help/. – Zugriffsdatum: 2023-07

[4] : *Computer Chess Engines: A Quick Guide*. – URL https://www.chess.com/article/view/computer-chess-engines. – Zugriffsdatum: 2023-07

[5] : *Forsyth-Edwards Notation*. – URL https://www.chessprogramming.org/Forsyth-Edwards_Notation. – Zugriffsdatum: 2023-07

[6] : *Forsyth-Edwards Notation (FEN)*. – URL https://www.chess.com/terms/fen-chess. – Zugriffsdatum: 2023-07

[7] : *Universal Chess Interface (UCI)*. – URL https://www.shredderchess.com/chess-features/uci-universal-chess-interface.html. – Zugriffsdatum: 2023-07

[8] Allis, L. V.: *Searching for Solutions in Games and Artificial Intelligence*. 1994. – URL http://fragrieu.free.fr/SearchingForSolutions.pdf. – Zugriffsdatum: 2023-07

[9] Canny, J.: A Computational Approach to Edge Detection. In: *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* PAMI-8 (1986), Nr. 6, S. 679–698. – URL https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.420.3300&rep=rep1&type=pdf. – Zugriffsdatum: 2023-07

[10] Danner, C. ; Kafafy, M.: Visual Chess Recognition. (2015). – URL https://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Danner_Kafafy.pdf. – Zugriffsdatum: 2023-07

[11] Ding, J.: ChessVision: Chess Board and Piece Recognition. (2016). – URL https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf. – Zugriffsdatum: 2023-07

[12] Dubrofsky, E.: Homography Estimation. (2009). – URL https://citeseerx.ist.psu.edu/doc/10.1.1.186.4411. – Zugriffsdatum: 2023-07

[13] Fei-Fei, L. u. a.: ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, URL https://ieeexplore.ieee.org/document/5206848. – Zugriffsdatum: 2023-07, 2009, S. 248–255

[14] Fei-Fei, L. u. a.: ImageNet Large Scale Visual Recognition Challenge. In: *CoRR* abs/1409.0575 (2014). – URL https://link.springer.com/content/pdf/10.1007/s11263-015-0816-y.pdf. – Zugriffsdatum: 2023-07

[15] Gershgorn, D.: *The data that transformed AI research—and possibly the world.* 2017-07-26. – URL https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world. – Zugriffsdatum: 2023-07

[16] Green, B.: *Canny Edge Detection Tutorial.* 2002. – URL https://web.archive.org/web/20160324173252/http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_weg22/can_tut.html. – Zugriffsdatum: 2023-07

[17] Hack, J. ; Ramakrishnan, P.: CVChess: Computer Vision Chess Analytics. (2015). – URL https://web.stanford.edu/class/cs231a/prev_projects_2015/chess.pdf. – Zugriffsdatum: 2023-07

[18] Hu, P. ; Luo, Y. ; Li, C.: Chinese Chess Recognition based on Projection Histogram of Polar Coordinates Image and FFT / Institute of Automation, Chinese Academy of Sciences, Beijing 100190, China. URL https://ieeexplore.ieee.org/abstract/document/5344001. – Zugriffsdatum: 2023-07, 2009. – Forschungsbericht

[19] Jian, C.: “凯撒杯”象棋锦标赛落幕中国小棋手夺得四项冠军. – URL https://game.huanqiu.com/article/9CaKrnJXdqu. – Zugriffsdatum: 2023-07

[20] Krishna, N.: *Camera Extrinsic Matrix with Example in Python*. 2022. – URL https://towardsdatascience.com/camera-extrinsic-matrix-with-example-in-python-cfe80acab8dd. – Zugriffsdatum: 2023-07

[21] Lin, T. u. a.: Focal Loss for Dense Object Detection / Facebook AI Research (FAIR). URL https://arxiv.org/pdf/1708.02002.pdf. – Zugriffsdatum: 2023-07, 2018. – Forschungsbericht

[22] Matas, J. ; Galambos, C. ; Kittler, J.: Robust Detection of Lines Using the Progressive Probabilistic Hough Transform. In: *Computer Vision and Image Understanding* 78 (2000), Nr. 1, S. 119–137. – URL https://www.sciencedirect.com/science/article/pii/S1077314299908317. – Zugriffsdatum: 2023-07. – ISSN 1077-3142

[23] Muller, B. R. ; Smith, W. A. P.: A Hierarchical Loss for Semantic Segmentation / Department of Computer Science, University of York, York, UK. URL https://www-users.york.ac.uk/~waps101/papers/muller2020hierarchical.pdf. – Zugriffsdatum: 2023-07, 2020. – Forschungsbericht

[24] OpenCV Team: *Basic concepts of the homography explained with code*. – URL https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html#lecture_16. – Zugriffsdatum: 2023-07

[25] OpenCV Team: *Camera Calibration*. – URL https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html. – Zugriffsdatum: 2023-07

[26] OpenCV Team: *Canny Edge Detection*. – URL https://docs.opencv.org/3.4/da/d22/tutorial_py_canny.html. – Zugriffsdatum: 2023-07

[27] OpenCV Team: *Contours: Getting Started*. – URL https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html. – Zugriffsdatum: 2023-07

[28] OpenCV Team: *Hough Line Transform*. – URL https://docs.opencv.org/3.4/d9/db0/tutorial_hough_lines.html. – Zugriffsdatum: 2023-07

[29] OpenCV Team: *Image Thresholding*. – URL https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html. – Zugriffsdatum: 2023-07

[30] Orémuš, Z.: *Chess Position Recognition from a Photo*, Masarykova Univerzita Fakulta Informatiky, Diplomarbeit, 2018. – URL https://is.muni.cz/th/meean/Master_Thesis.pdf. – Zugriffsdatum: 2023-07

[31] O'Shea, K. ; Nash, R.: *An Introduction to Convolutional Neural Networks*. 2015. – URL https://arxiv.org/pdf/1511.08458.pdf. – Zugriffsdatum: 2023-07

[32] Park, D.: Space-state complexity of Korean chess and Chinese chess. (2015). – URL https://arxiv.org/ftp/arxiv/papers/1507/1507.06401.pdf. – Zugriffsdatum: 2023-07

[33] Poiker, T. K. ; Douglas, D. H.: Reflection Essay: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature, URL https://www.semanticscholar.org/paper/Reflection-Essay%3A-Algorithms-for-the-Reduction-of-a-Poiker-Douglas/e46ac802d7207e0e51b5333456a3f46519c2f92d. – Zugriffsdatum: 2023-07, 1973

[34] Simonyan, K. ; Zisserman, A.: Very Deep Convolutional Networks for Large-Scale Image Recognition / Visual Geometry Group, Department of Engineering Science, University of Oxford. URL https://arxiv.org/pdf/1409.1556.pdf. – Zugriffsdatum: 2023-07, 2014. – Forschungsbericht

[35] Stapczynski, C.: *History of Chess | From Early Stages to Magnus*. 2023-03-23. – URL https://www.chess.com/article/view/history-of-chess. – Zugriffsdatum: 2023-07

[36] World Xiangqi Federation ; Chinese Xiangqi Association: *World XiangQi Rules*. People's Sports Publishing House, 2018. – URL https://www.wxf-xiangqi.org/images/wxf-rules/2018_World_XiangQi_Rules_English2018.pdf. – Zugriffsdatum: 2023-07

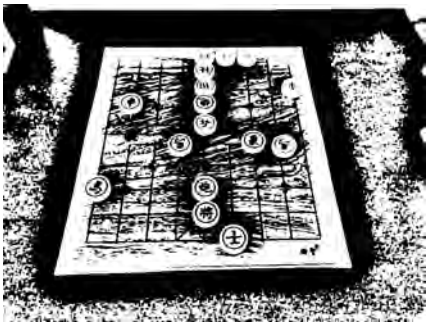
[37] Zhang, Z.: A flexible new technique for camera calibration. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), Nr. 11, S. 1330–1334. – URL https://ieeexplore.ieee.org/document/888718. – Zugriffsdatum: 2023-07

A Anhang

Abb. A.1 und A.2 (nächste zwei Seiten) zeigen die schrittweise Abfolge des Programms. Ausschnitte sind der Einfachheit halber reihenweise dargestellt. Intern sind die Ausschnitte einzeln. Der FEN-String ist hierbei `3a5/4k4/4c3n/9/1Nb2N3/4P4/4c2r1/R3B4/4A4/2BAK4 b`, der beste Zug für schwarz (weil schwarz unten ist) `h3h0` (hier Matt in 1). Die Ausführungsdauer beträgt etwa 22 Sekunden, wobei den Großteil die Klassifizierung mit dem neuronalen Netz ausmacht.

(a) Ausgangsbild

(b) Graustufen, Gaussscher Filter



(c) Adaptive Thresholding

(d) Konturen

(e) Approximation/Eckpunkte

Abbildung A.1: Kompletter Ablauf des Programms in Einzelschritten Teil 1

(a) Entzerrtes Brett

(b) Erweiterung um Rand

(c) Ausschnitt Reihe 1

(d) Ausschnitt Reihe 2

(e) Ausschnitt Reihe 3

(f) Ausschnitt Reihe 4

(g) Ausschnitt Reihe 5

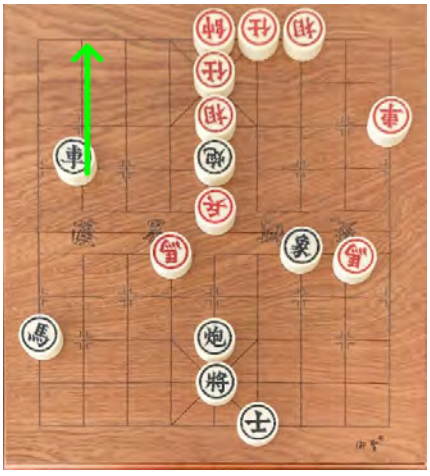
(h) Ausschnitt Reihe 6

(i) Ausschnitt Reihe 7

(j) Ausschnitt Reihe 8

(k) Ausschnitt Reihe 9

(l) Ausschnitt Reihe 10



(m) Darstellung bester Zug

Abbildung A.2: Kompletter Ablauf des Programms in Einzelschritten Teil 2

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original