

PROJEKTDOKUMENTATION

Bildbasierte digitale Aufzeichnung von realen
Schach-Spielen

vorgelegt an der TH Köln
Campus Gummersbach
im Studiengang
Medieninformatik Master

ausgearbeitet von:
AILEEN JURKOSEK (11134311)
JULIAN HARDTUNG (11104591)

Prüfer: Matthias Groß

Gummersbach, im Februar 2023

Inhaltsverzeichnis

1 Einleitung	1
2 Grundlage	2
2.1 Konfiguration von Schachspielen	2
2.2 Board Localisation	2
2.3 Occupancy Classification	5
2.4 Piece Classification	6
3 Projekt	7
3.1 Umsetzung der Board Localisation	7
3.2 Umsetzung der Occupancy Classification	15
3.3 Umsetzung der Piece Classification	18
4 Fazit	21
Abbildungsverzeichnis	22
Literaturverzeichnis	23

1 Einleitung

Die digitale Aufzeichnung von Schachspielen hat in der Schachgemeinschaft einen hohen Stellenwert durch die Möglichkeit zur Analyse von Spielen oder dem Erstellen von Trainingseinheiten. Trotz zunehmender Digitalisierung und der Verfügbarkeit von Online-Schachspielen wird häufig auf physische Schachbretter und -figuren für Spiele zurückgegriffen. Durch den hohen Aufwand einer manuellen Digitalisierung solcher realen Schachspiele ist eine Automatisierung dieses Prozesses erstrebenswert.

Die folgende Projektarbeit im Modul „Bildbasierte Computergrafik“ im Wintersemester 2022/23 behandelt die digitale Aufzeichnung einzelner Spielpositionen eines physisch stattfindenden Schachspiels. Hierbei sollen mittels Bildverarbeitungsalgorithmen Schachbretter und die darauf befindlichen Schachfiguren automatisch erkannt werden, wodurch die digitale Aufzeichnung von Schachspielen einfacher und effizienter gestaltet werden kann.

Die Arbeit baut auf dem in der Seminarphase behandelten Paper „Determining Chess Game State From an Image“ der Autoren Georg Wölflein und Ognjen Arandjelović auf. Darin wird ein Algorithmus behandelt, welcher die Schritte der *Board Localisation*, *Occupancy Classification* und *Piece Classification* umfasst. Die *Board Localisation* bezieht sich hierbei auf das Erkennen eines Schachbrettes auf einem Foto. Die *Occupancy Classification* stellt fest, welche Felder eines Schachbrettes belegt sind. Mit der *Piece Classification* werden die einzelnen Schachfiguren und ihre jeweilige Position auf dem Brett erkannt. Im Rahmen dieses Projekts sollen die einzelnen Schritte eigenständig implementiert und getestet werden. Mit der Umsetzung dieses Projekts wird somit das Verständnis einzelner Bildverarbeitungsalgorithmen vertieft.

2 Grundlage

Auf Basis des Papers „*Determining Chess Game State From an Image*“ von Georg Wöllein und Ognjen Arandjelović und dem im Zuge dessen entstandenen Codes wird die Grundlage für das Projekt gebildet. Diesbezüglich werden nachfolgend der Hintergrund des Prozesses, sowie die einzelnen Schritte der *Board Localisation*, *Occupancy Classification* und *Piece Classification* erläutert.

2.1 Konfiguration von Schachspielen

Die Konfiguration von Schachspielen bietet eine wichtige Hilfe für Amateur Schachspieler, um Spielzüge besser nachzuvollziehen und sich insgesamt zu verbessern. Dabei ist es hilfreich, Spielpositionen nach einem Spiel an einem Computer zu analysieren. Findet das Spiel hierbei nicht digital, sondern an einem realen Schachbrett statt, so wäre es notwendig, dass Spieler einzelne spezifische Spielposition durch Fotos festhalten und die einzelnen Positionen der Spielfiguren händisch am Computer eintragen. Dieser Vorgang ist zum einen aufwändig und zum anderen fehleranfällig. Ziel ist daher eine Automatisierung, durch die ein Foto einer Schachposition auf ein Format gemappt wird, welches kompatibel für eine Schach Engine ist und jede Spielposition dadurch digital dargestellt werden kann.

2.2 Board Localisation

Die *Board Localisation* beschreibt den Vorgang, bei welchem das Schachbrett digital erkannt wird. Eingeteilt wird dabei in das Finden der Schnittpunkte und die Berechnung der Homografie. Für den ersten Schritt werden in einem Bild horizontale und vertikale Linien und anschließend die jeweiligen Schnittpunkte identifiziert, um das Schachbrettmuster erkennen zu können. Dazu wird das Bild zunächst in ein Grauwertbild konvertiert, worauf anschließend ein *Canny Edge Detector* angewendet wird. Um die einzelnen Linien im Bild zu finden, welche durch die zuvor gefundenen Ecken geformt werden, wird anschließend eine *Hough Transformation* durchgeführt. Gefunden werden dabei durchschnittlich 200 Linien, entweder horizontal oder vertikal. Doppelte oder sich stark ähnelnde Linien werden dabei durch die Anwendung eines agglomerativen Cluster Algorithmus entfernt. Jede Linie beginnt dabei mit einem eigenen Cluster, wobei Clusterpaare zusammengeführt werden (siehe Abbildung 1).

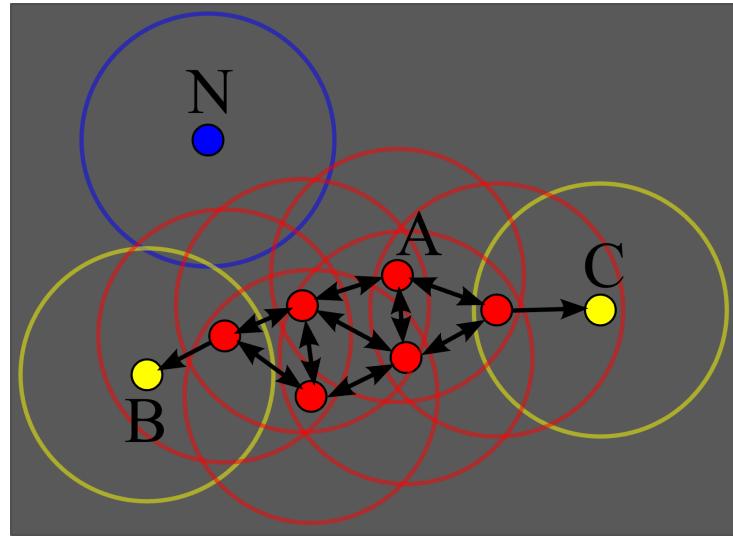


Abbildung 1: Visualisierung des Cluster-Algorithmus

Der kleinste Winkel zwischen zwei gegebenen Linien dient hierbei als Abstandsmetrik. Der mittlere Winkel der beiden oberen Cluster bestimmt, welcher Cluster repräsentativ die jeweilige vertikale und horizontale Linie darstellt. Nach Beseitigung der übrigen Linien werden die Schnittpunkte der horizontalen Linien mit der mittleren vertikalen Linie gefunden (siehe Abbildung 2). Ähnliche Linien werden hierbei durch die Verwendung des DBSCAN-Clusters gruppiert (siehe Abbildung 3). Das gleiche Verfahren wird bei den senkrechten Linien durchgeführt (siehe Wölflein u. Arandjelović, 2021c, S.4).

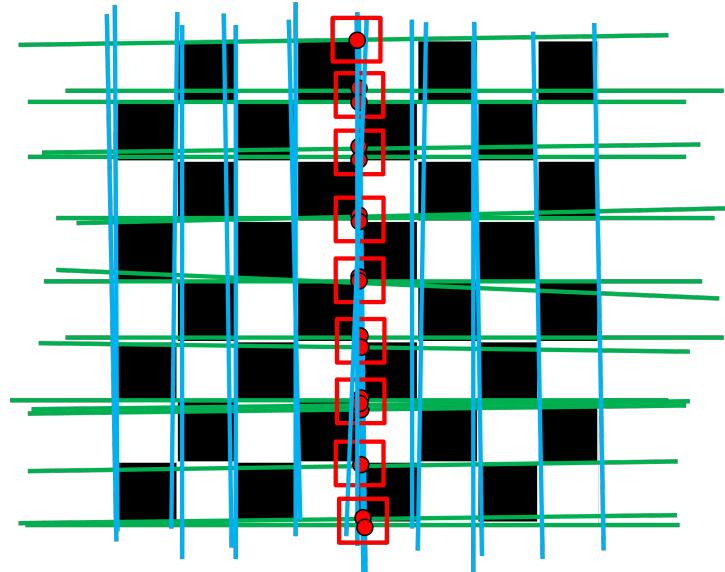


Abbildung 2: Finden vertikaler und horizontaler Linien mittels DBSCAN

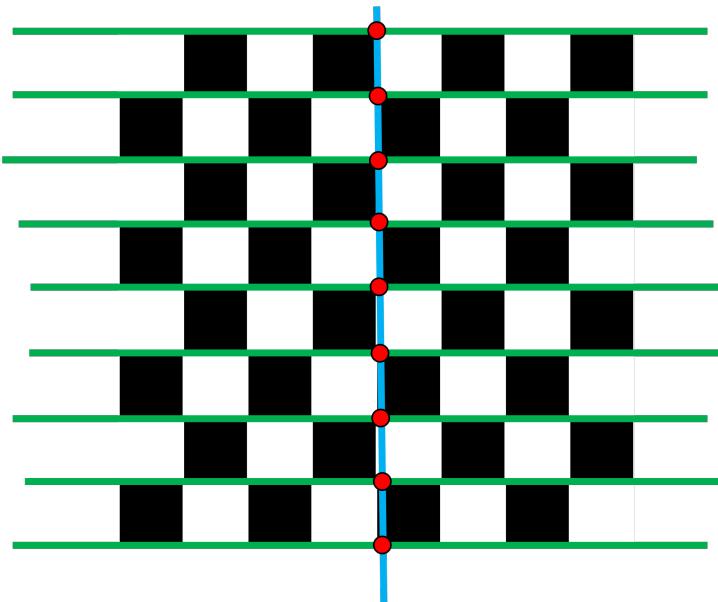


Abbildung 3: Eliminieren gleicher Linien

Die Berechnung der Homografie wird durchgeführt, wenn weniger als neun horizontale und vertikale Linien erkannt werden. Dabei wird ein *Warping* des Eingabebildes vorgenommen, sodass die Schnittpunkte ein regelmäßiges Raster von Quadranten bilden. Anschließend wird eine projektive Transformation durchgeführt, welche durch eine Homografie-Matrix H charakterisiert wird. Um die Homografie zu berechnen, wird nun ein RANSAC-basierter Algorithmus angewandt. Dazu werden vier Schnittpunkte als Sample verwendet. Mit der Homografie-Matrix H werden die vier Schnittpunkte auf ein Rechteck mit der Breite $s_x = 1$ und $s_y = 1$ abgebildet. Alle Schnittpunkte werden anschließend mit H projiziert, wonach die Anzahl der Inliner gezählt werden. Falls die Größe des Inliner-Sets größer ist als die der vorherigen Iteration, werden das Inliner-Set und die Homografie-Matrix beibehalten. Der Vorgang wird ab Schritt 1 wiederholt, bis mindestens die Hälfte der Schnittpunkte Inliner sind. Die Lösung mit den wenigsten Error-Feldern wird unter der Verwendung der identifizierten Inliner auf die Homografie-Matrix angewandt (siehe Wölflein u. Arandjelović, 2021c, S.5).

2.3 Occupancy Classification

Bevor die *Piece Classification* durchgeführt werden kann, wird die *Occupancy Classification* vollzogen, um eine große Anzahl von falsch positiven Erkennungen zu vermeiden. Dabei wird ein *Binary Classifier* darauf trainiert zu erkennen, ob ein Feld des Schachbrettes leer oder belegt ist (siehe Abbildung 4). Dazu werden sechs *Convolutional Neural Network (CNN)* Architekturen genutzt, wovon zwei 100 x 100 Pixel Input Bilder akzeptieren und vier 50 x 50 Pixel Input Bilder einlesen. Die einzelnen Bilder unterscheiden sich in der Anzahl verschiedener Schichten wie Faltungsschichten, Pooling Schichten und vollständig verbundenen Schichten. Modelle werden dann durch 4er-Tupel beschrieben, welche die drei genannten Schichten und die Seitenlänge des jeweiligen Input-Bildes beinhalten. Die im Modell final verbundenen Schichten enthalten anschließend zwei Ausgabeeinheiten, welche die beiden Klassen leer und belegt repräsentieren. Um ein zuverlässigeres Ergebnis gewährleisten zu können, kann ein Optimizer genutzt oder ein Pre-Training mit einem weiteren Dataset durchgeführt werden. (siehe Wölflein u. Arandjelović, 2021c, S.7)



Abbildung 4: Occupancy Classification (siehe Wölflein u. Arandjelović, 2021c, S.7)

2.4 Piece Classification

In einem Schachspiel gibt es nun verschiedene Figuren, jeweils in den Farben schwarz und weiß, welche voneinander unterschieden werden müssen: Bauer, Springer, Läufer, Turm, Dame und König. Um eine Figur in einem Bild erkennen zu können, wird die *Piece Classification* durchgeführt. Dabei wird ein ausgeschnittenes Bild eines belegten Feldes als Eingabe an den Algorithmus gegeben und die Schachfigur auf diesem entsprechenden Feld als Ausgabe geliefert. Dabei muss darauf geachtet werden, dass die zu identifizierende Figur eindeutig zu erkennen ist. Eine Heuristik, welche dafür sorgt abgeschnittene Figuren zu vermeiden, sieht vor, dass die Höhe und Breite der *Bounding Boxes* des jeweiligen Feldes erweitert werden (siehe Abbildung 5). Anschließend werden die *Bounding Boxes* ggf. gedreht, sodass sich das betroffene Feld mit seiner Figur in der unteren linken Ecke des Bildes befindet. Dieser Schritt hilft dem *Classifier* zu erkennen, welcher Bildbereich verarbeitet werden soll, wodurch eine eindeutige Erkennung möglich ist. (siehe Wölflein u. Arandjelović, 2021c, S.8)

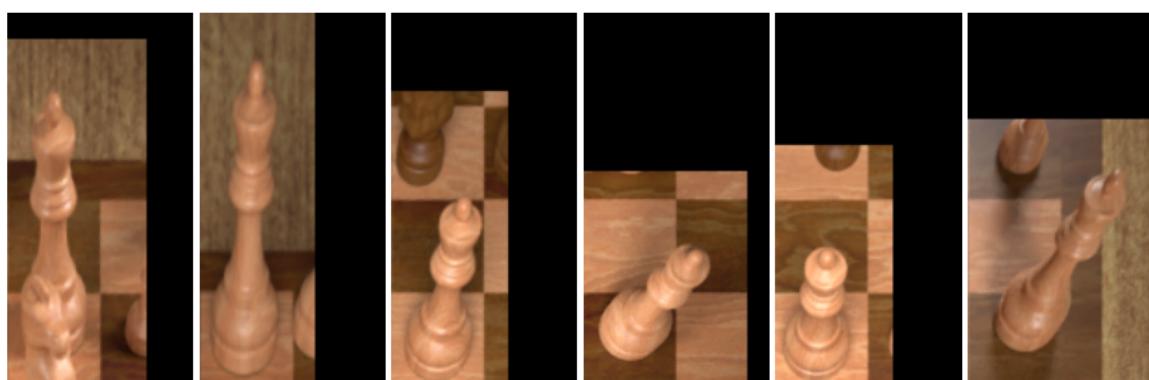


Abbildung 5: Piece Classification (siehe Wölflein u. Arandjelović, 2021c, S.9)

3 Projekt

Im Projekt stand nun die eigene Umsetzung der einzelnen Schritte der Board Localisation, der Occupancy Classification und der Piece Classification im Fokus. Orientiert wurde sich hierbei am bestehenden Code, welcher von den Autoren des behandelten Papers zur freien Verfügung steht (siehe Wölflein u. Arandjelović (2021a)). Des Weiteren wurde der von den Entwicklern verwendete Datensatz (siehe Wölflein u. Arandjelović (2021b)) für die eigene Implementierung verwendet. Entwickelt wurden jeweils verschiedene Ansätze für die einzelnen Schritte des Algorithmus. Diese galten hierbei allerdings zunächst als Einstieg und umfassten bisher teils keine Integration neuronaler Netze, wie sie im Paper diskutiert wurden, und des gegebenen Datensatzes.

3.1 Umsetzung der Board Localisation

Bei der Umsetzung der Board Localisation wurden verschiedene Ansätze gefunden. An sprechend waren hierbei die OpenCV-eigenen Funktionen `findChessboardCorners()` (siehe OpenCV-Dokumentation (2022e)) und `drawChessboardCorners()` (siehe OpenCV-Dokumentation (2022c)). Diese sind Teil des Moduls „Camera Calibration and 3D Reconstruction“ (siehe OpenCV-Dokumentation (2022a)) und können für Teile der Board Localisation genutzt werden (siehe Abbildung 6 und Abbildung 7).

```
# Load image
image = cv2.imread("chessboard.png")

# Convert image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect chessboard corners
found, corners = cv2.findChessboardCorners(gray, (7,7), None)
```

Abbildung 6: Anwendung der `findChessboardCorners()`-Methode

```
# Draw chessboard corners on the image if found
if found:
    cv2.drawChessboardCorners(image, (7,7), corners, found)
```

Abbildung 7: Anwendung der `drawChessboardCorners()`-Methode

Dieser zunächst gewählte einfache Ansatz lieferte allerdings auch nach Warping eines eingelesenen Schachbrett-Bildes keine Eckpunkte als Ausgabe zurück.

Ein alternativer Ansatz umfasst den Einsatz der bereits bestehenden OpenCV Methode `goodFeaturesToTrack()` (siehe OpenCV-Dokumentation (2022g)) aus dem Modul Feature Detection (siehe OpenCV-Dokumentation (2022d)). Diese Funktion dient dazu, auffällige Ecken in einem eingelesenen Bild zu identifizieren. Als Parameter erhält die Funktion hierbei das in ein Grauwertbild konvertierte Eingabebild, das maximale Level der auszugebenden Ecken im Bild, das Qualitätslevel der zu findenden Ecken und die minimale Distanz zwischen den zu findenden Ecken. Als optionale Parameter können hierbei zusätzlich ein Array mit den gefundenen Ecken und eine Maske, die bestimmte Bereiche des Bildes markiert, in denen nicht nach Merkmalen gesucht werden soll, angegeben werden, sowie ob ein Harris Detector zum Finden der Punkte angewendet werden soll und ein entsprechender Parameter für diesen Detector (siehe Abbildung 8).

```
# Define the parameters for goodFeaturesToTrack
max_corners = 200
quality_level = 0.001
min_distance = 20
block_size = 3
use_harris_detector = True
k = 0.04

# Apply goodFeaturesToTrack to find corners
corners = cv2.goodFeaturesToTrack(gray, max_corners, quality_level, min_distance,
| | | | | None, None, block_size, use_harris_detector, k)
```

Abbildung 8: Anwendung der `goodFeaturesToTrack()`-Methode

Durch die Variation der einzelnen Parameter wie der Anzahl der maximalen Ecken, des Qualitäts-Level und der minimalen Distanz kann bestimmt werden, wie viele Ecken erkannt werden sollen. Mit dem obigen Code können hierbei die in Abbildung 9 und Abbildung 10 zu sehenden Ausgaben erzielt werden.

Zu erkennen ist, dass sowohl in Abbildung 9 als auch in Abbildung 10 zwar fast alle Ecken des Schachbrettes erkannt werden, jedoch auch Features außerhalb des Schachbrettes vom Algorithmus gefunden werden. Beim Originalbild ist dies darauf zurückzuführen, dass der Untergrund des Schachbrettes eine leichte Struktur aufweist, in der Ecken erkannt werden können. Im entzerrten Bild ist dies durch die leichte farbliche Anpassung nicht mehr der Fall, allerdings entsteht durch das Entzerren eine neue Kante, welche hierbei ebenfalls als Struktur mit Ecken aufgefasst wird.

Die Funktion `findHomography()` (siehe OpenCV-Dokumentation (2022f)) kann nun verwendet werden, um dem entgegen zu wirken. Sie bewirkt das Beseitigen perspektivischer Verzerrungen und kann somit ein genaueres Ergebnis gewährleisten (siehe Abbildung 11). Allerdings kann auch mit dieser Ergänzung keine Ausgabe erzielt werden.

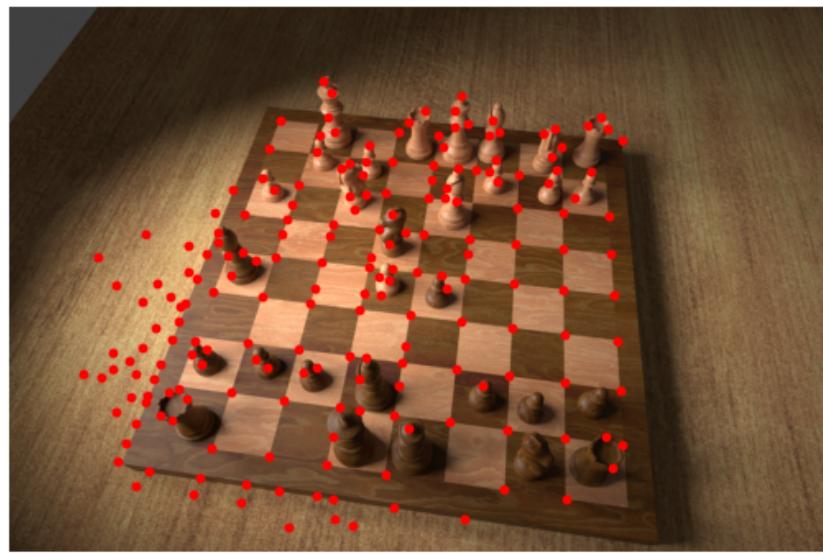


Abbildung 9: Ausgabe vor Entzerrung des Bildes

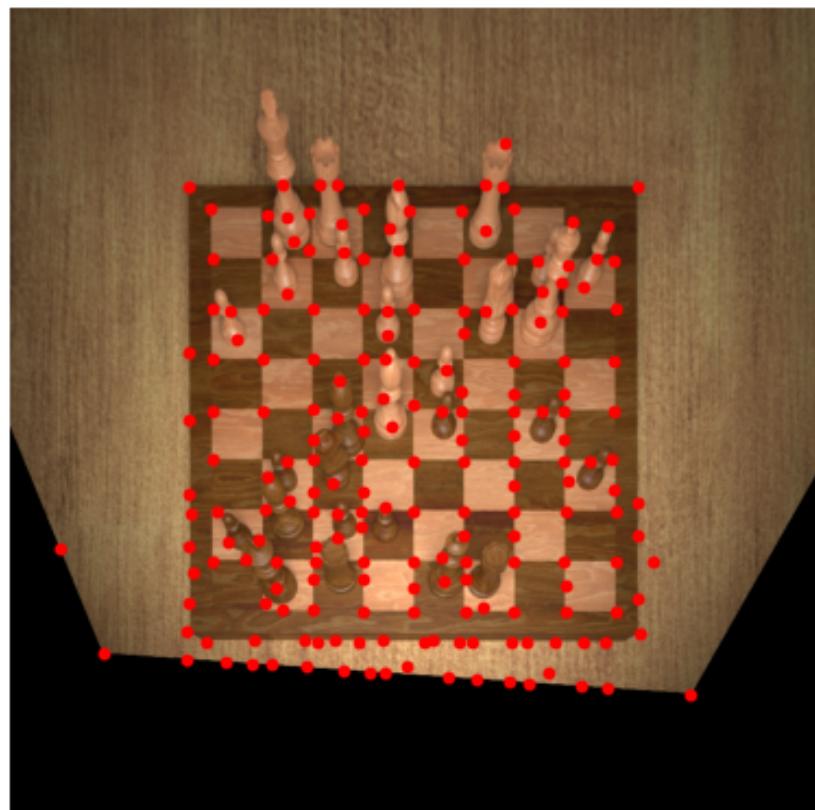


Abbildung 10: Ausgabe nach Entzerrung des Bildes

Ein weiterer Ansatz verfolgte die Verwendung der OpenCV Funktionen `Canny()` (siehe OpenCV-Dokumentation (2022b)) und `HoughLines()` (siehe OpenCV-Dokumentation (2022h)), zur vereinfachten Anwendung des Canny-Edge-Detectors sowie der Hough-Transformation zum Finden der horizontalen und vertikalen Linien im Bild und der anschließenden Identifizierung der Eckpunkte (siehe Abbildung 12).

```

# Find homography matrix using the corner points and the pattern points
H, _ = cv2.findHomography(corners, pattern_points)

# Warp the image to remove perspective distortion
result = cv2.warpPerspective(image, H, (image.shape[1], image.shape[0]))

```

Abbildung 11: Anwendung der `findHomography()`-Methode

```

# Apply Canny edge detection
edges = cv2.Canny(gray, 100, 200)

# Find Hough lines
lines = cv2.HoughLines(edges, 1, np.pi/180, 100)

# Find intersection points of Hough lines
intersections = []
for line1 in lines:
    rho1, theta1 = line1[0]
    for line2 in lines:
        rho2, theta2 = line2[0]
        if abs(theta1 - theta2) > np.pi/4: # Only consider orthogonal lines
            x, y = np.array([np.cos(theta1), np.sin(theta1)]), np.array([np.cos(theta2), np.sin(theta2)])
            intersection = np.int32(np.round(np.linalg.solve(np.vstack([x, y]), np.array([rho1, rho2]))))
            if intersection[0] >= 0 and intersection[1] >= 0 and intersection[0] < gray.shape[1] and intersection[1] < gray.shape[0]:
                intersections.append(intersection)

# Find the four corners
corner_candidates = np.array(intersections, dtype=np.int32)
distances = cv2.distanceTransform(255 - edges, cv2.DIST_L2, cv2.DIST_MASK_PRECISE)
corner_scores = distances[corner_candidates[:, 1], corner_candidates[:, 0]]
corner_ids = np.argsort(corner_scores)[:4]
corners = corner_candidates[corner_ids]

return corners

```

Abbildung 12: Anwendung des Canny-Edge-Detectors und der Hough Transformation

Der Code liefert die in Abbildung 13 und Abbildung 14 gezeigten Ergebnisse. Hierbei werden zwar 4 Punkte angezeigt, diese zeigen aber nicht die Ecken des Schachbrettes, sondern 4 zusammenhanglose Ecken auf dem Bild, welche teilweise übereinander liegen.

Da die erprobten Ansätze mit ihren angewandten OpenCV-eigenen Funktionen teils nicht im gewünschten Maße zusammenwirken, wird sich für den weiteren Verlauf am Code orientiert, welcher im Zuge des behandelten Papers entstanden ist. Hierbei wurden verschiedene Änderungen vorgenommen, um ein lauffähiges System zu erzeugen. Die Board Localization stellt einen wichtigen Teil der Schachspielerkennung dar, da sie die Grundlage für die weiteren folgenden Erkennungen darstellt. Deswegen wird diese sehr detailliert durchgeführt, da mit einer simplen Umsetzung keine genauen Ergebnisse erzielt werden konnten. Die einzelnen umgesetzten Schritte, welche sich an den Ergebnissen des Papers orientieren, werden folgend beschrieben:

Das Finden der Ecken des Schachbrettes beinhaltet mehrere Bestandteile. Zunächst wird die Größe des Bildes angepasst, damit die gleiche Ausgangslage für die einzelnen eingelesenen Bilder besteht und die Bilder ggf. schneller verarbeitet werden können. Der für das

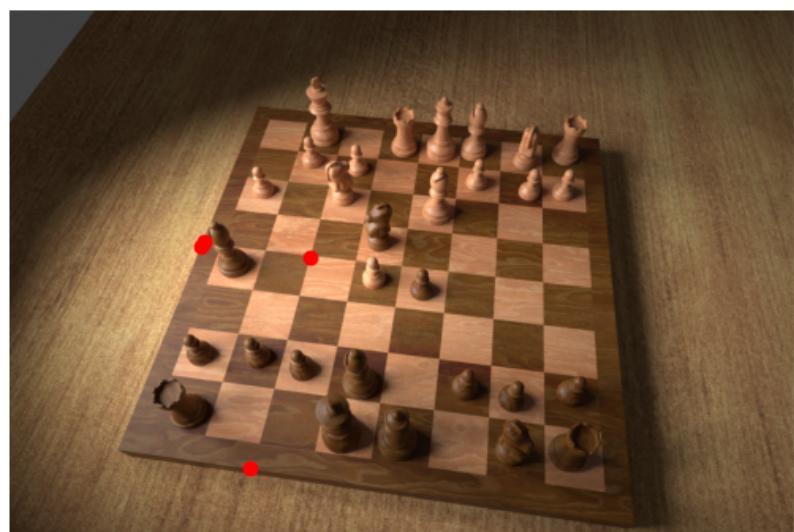


Abbildung 13: Ausgabe vor Warping des Bildes nach Hough Transformation

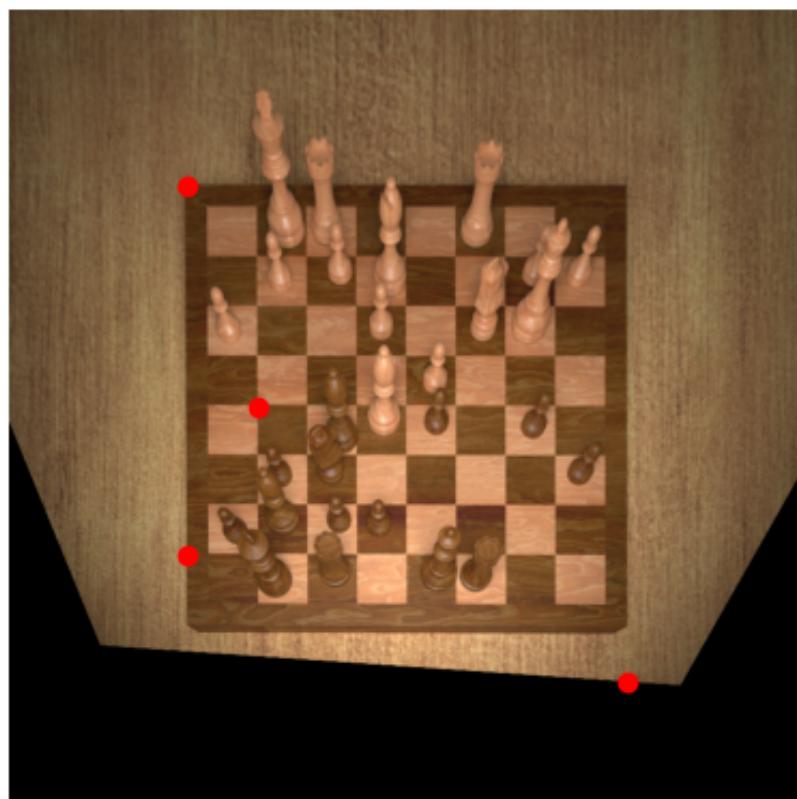


Abbildung 14: Ausgabe nach Warping des Bildes nach Hough Transformation

Resizing festgelegte Wert ist hierbei in der Config-Datei hinterlegt (siehe Abbildung 15). Darin werden weitere Parameter angepasst, wie verschiedene Threshold-Parameter. Diese geben z.B. an, welche Helligkeiten in den Bildern erkannt werden sollen. Auch der Anteil der möglichen Ausreißer wird hier für den späteren Verlauf festgelegt.

```

RESIZE_IMAGE_WIDTH = 1200
EDGE_DETECTION_LOW_THRESHOLD = 120
EDGE_DETECTION_HIGH_THRESHOLD = 300
EDGE_DETECTION_APERTURE = 3

LINE_DETECTION_THRESHOLD = 100
LINE_DETECTION_DIAGONAL_LINE_ELIMINATION = True
LINE_DETECTION_DIAGONAL_LINE_ELIMINATION_THRESHOLD_DEGREES = 30

MAX_OUTLIER_INTERSECTION_POINT_RATIO_PER_LINE = .5

RANSAC_OFFSET_TOLERANCE = .15
RANSAC_BEST SOLUTION_TOLERANCE = .1

BORDER_REFINEMENT_NUM_SURROUNDING_SQUARES_IN_WARPED_IMG = 5
BORDER_REFINEMENT_WARPED_SQUARE_SIZE = [50, 50]
BORDER_REFINEMENT_SOBEL_KERNEL_SIZE = 3
BORDER_REFINEMENT_LINE_WIDTH = 4

```

Abbildung 15: Definierte Config-Datei

Nach dem Resizing wird das Bild anschließend in ein Grauwertbild umgewandelt, wonach mit dem Canny-Edge-Detector in der Methode `detectEdges()` (siehe Abbildung 16) alle Ecken und Kanten im Bild identifiziert werden können. Mithilfe der gefundenen Kanten werden dann durch die Anwendung der Hough-Transformation in der Funktion `detectLines()` (siehe Abbildung 17) die Linien, welche das Muster des Schachbrettes symbolisieren, erkannt.

```

def detect_edges(grey: np.ndarray) -> np.ndarray:
    if grey.dtype != np.uint8:
        grey = grey / grey.max() * 255
        grey = grey.astype(np.uint8)
    edges = cv2.Canny(grey,
                      cfg.EDGE_DETECTION_LOW_THRESHOLD,
                      cfg.EDGE_DETECTION_HIGH_THRESHOLD,
                      cfg.EDGE_DETECTION_APERTURE)

    return edges

```

Abbildung 16: Definition der `detectEdges()`-Methode

Im Anschluss daran werden die vertikalen und horizontalen Linien geclustert und sortiert, damit doppelte Linien eliminiert werden können. Nach dem Eliminieren gleicher Linien werden die Schnittpunkte zwischen den übrigen Linien mit `getIntersectionPoints()` ermittelt. Die Methode erhält hierbei die horizontalen und vertikalen Linien als Eingabe und liefert eine Matrix zurück, welche die Anzahl der horizontalen und vertikalen Linien, sowie die Koordinaten der entsprechenden Schnittpunkte enthält.

```

def detect_lines(edges: np.ndarray) -> np.ndarray:
    lines = cv2.HoughLines(edges, 1, np.pi/360, cfg.LINE_DETECTION_THRESHOLD)
    lines = lines.squeeze(axis=-2)
    lines = fix_negative_rho_in_hesse_normal_form(lines)

    if cfg.LINE_DETECTION_DIAGONAL_LINE_ELIMINATION:
        threshold = np.deg2rad(
            cfg.LINE_DETECTION_DIAGONAL_LINE_ELIMINATION_THRESHOLD_DEGREES)
        vmask = np.abs(lines[:, 1]) < threshold
        hmask = np.abs(lines[:, 1] - np.pi / 2) < threshold
        mask = vmask | hmask
        lines = lines[mask]
    return lines

```

Abbildung 17: Definition der `detectLines()`-Methode

Um die vier besten Eckpunkte des Schachbrettes finden zu können, wird nun ein RANSAC-Algorithmus verwendet (siehe Abbildung 18). Über eine Schleife werden hierbei vier zufällige zuvor berechnete Schnittpunkte gewählt, mit welchen dann eine Homografie-Matrix berechnet werden kann. Diese Matrix wird nun auf alle gefundenen Schnittpunkte angewandt, wodurch berechnet wird, welche Punkte als Inliner angesehen werden können. Inliner sind hierbei genau die Punkte, die im Quadrat liegen, welches von den vier aktuell verwendeten Punkten geformt wird. Die Schleife wird beendet, wenn entweder 200 Iterationen durchgeführt wurden oder ein Schwellenwert von 30 gefundenen Inlinern überschritten wird. Die Konfiguration mit den meisten Inlinern wird ausgewählt, da mit der daraus resultierenden Homografie-Matrix die zuverlässigsten Eckpunkte des Schachbrettes identifiziert werden können.

```

while iterations < 200 or best_num_inliers < 30:
    row1, row2 = choose_from_range(len(horizontal_lines))
    col1, col2 = choose_from_range(len(vertical_lines))
    transformation_matrix = compute_homography(all_intersection_points,
                                                row1, row2, col1, col2)
    warped_points = warp_points(
        transformation_matrix, all_intersection_points)
    warped_points, intersection_points, horizontal_scale, vertical_scale = discard_outliers(warped_points, all_intersection_points)
    num_inliers = np.prod(warped_points.shape[:-1])
    if num_inliers > best_num_inliers:
        warped_points *= np.array((horizontal_scale, vertical_scale))

    # Quantize and reject duplicates
    (xmin, xmax, ymin, ymax), scale, quantized_points, intersection_points, warped_img_size = configuration = quantize_points(warped_points, intersection_points)

    # Calculate remaining number of inliers
    num_inliers = np.prod(quantized_points.shape[:-1])

    if num_inliers > best_num_inliers:
        best_num_inliers = num_inliers
        best_configuration = configuration
    iterations += 1
    if iterations > 10000:
        raise Exception("RANSAC produced no viable results")

```

Abbildung 18: RANSAC-Schleife

Mithilfe der Homografie-Matrix können nun die homogenen Koordinaten der Schnittpunkte berechnet werden, wodurch die Perspektive des Bildes anschließend angepasst werden kann, sodass ein gerades Raster entsteht (siehe Abbildung 19). Die vier Eckpunkte werden zuletzt in der ursprünglichen Bildperspektive berechnet und zurückgegeben, wodurch sich zudem die horizontalen und vertikalen Grenzen des Schachbrettes ergeben.

```

# Retrieve best configuration
(xmin, xmax, ymin, ymax), scale, quantized_points, intersection_points, warped_img_size = best_configuration

# Recompute transformation matrix based on all inliers
transformation_matrix = compute_transformation_matrix(
    | intersection_points, quantized_points)
inverse_transformation_matrix = np.linalg.inv(transformation_matrix)

# Warp grayscale image
dims = tuple(warped_img_size.astype(int))
warped = cv2.warpPerspective(grey_img, transformation_matrix, dims)
borders = np.zeros_like(grey_img)
borders[3:-3, 3:-3] = 1
warped_borders = cv2.warpPerspective(borders, transformation_matrix, dims)
warped_mask = warped_borders == 1

# Refine board boundaries
xmin, xmax = compute_vertical_borders(warped, warped_mask, scale, xmin, xmax)
scaled_xmin, scaled_xmax = (int(x * scale[0]) for x in (xmin, xmax))
warped_mask[:, :scaled_xmin] = warped_mask[:, scaled_xmax:] = False
ymin, ymax = compute_horizontal_borders(warped, warped_mask, scale, ymin, ymax)

# Transform boundaries to image space
corners = np.array([[xmin, ymin],
                    [xmax, ymin],
                    [xmax, ymax],
                    [xmin, ymax]]).astype(float)
corners = corners * scale
img_corners = warp_points(inverse_transformation_matrix, corners)
img_corners = img_corners / img_scale
return sort_corner_points(img_corners)

```

Abbildung 19: Berechnung der Homografie und der finalen Eckpunkte

Mithilfe des überarbeiteten Codes kann die in Abbildung 20 gezeigte finale Ausgabe erzielt werden, welche nun nur noch die vier Eckpunkte des Bildes erkennt und diese auf dem Bild einzeichnet:



Abbildung 20: Ausgabe der fertigen Board Localisation

3.2 Umsetzung der Occupancy Classification

Auch für die Umsetzung der *Occupancy Classification* wurde sich an den Ergebnissen nach Wölflein u. Arandjelović orientiert. Für die Durchführung wurden auf das in der *Board Localisation* erkannte Schachbrett mehrere CNNs mit unterschiedlichen Parametern trainiert, um die belegten und freien Schachfelder zu finden. Durch das Training verschiedener CNNs konnte überprüft werden, ob es zu unterschiedlichen Genauigkeiten bei der Einstellung verschiedener Parameter kommt.

Der in Abbildung 21 gegebene Codeblock zeigt die schrittweise Erkennung der belegten Spielfelder. So werden als Input zunächst vier Werte benötigt:

1. `self` -> enthält die Occupancy und Piece Classifier, sowie weitere Konfigurationsparameter
2. `img: np.ndarray` -> die originale Bildaufnahme des Schachbretts
3. `turn: chess.Color` -> aus welcher Sicht das Schachbrett aufgenommen wurde
4. `corners: np.ndarray` -> die Pixelpositionen der Eckpunkte des Schachbretts

```
def _classify_occupancy(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray) -> np.ndarray:  
  
    # warp image to regular grid  
    warped = create_occupancy_dataset.warp_chessboard_image(img, corners)  
  
    # cut out squares with margin around them  
    square_imgs = map(functools.partial(create_occupancy_dataset.crop_square, warped, turn=turn), self._squares)  
    square_imgs = map(Im2darray.fromarray, square_imgs)  
    square_imgs = map(self._occupancy_transforms, square_imgs)  
    square_imgs = list(square_imgs)  
    square_imgs = torch.stack(square_imgs)  
    square_imgs = device(square_imgs)  
  
    # classify if a square is occupied or not  
    occupancy = self._occupancy_model(square_imgs)  
    occupancy = occupancy.argmax(axis=-1) == self._occupancy_cfg.DATASET.CLASSES.index("occupied")  
    occupancy = occupancy.cpu().numpy()  
  
    return occupancy
```

Abbildung 21: Code der Occupancy Classification

Mit diesen Werten kann nun die *Occupancy Classification* durchgeführt werden. Im ersten Schritt wird hier das Schachbrett mit einer projektiven Transformation entzerrt, sodass die einzelnen Spielfelder quadratisch sind und in einem regelmäßigen Raster liegen (siehe Abbildung 22). Dies ist wichtig für das nachfolgende Auseinanderschneiden der einzelnen Felder, welches wiederum die Basis für die eigentliche *Occupancy Classification* darstellt.



(a) Unbearbeitetes Eingabebild



(b) Entzerrtes Bild des Schachbrettes

Abbildung 22: Entzerrung des Schachbrettes

In diesem Schritt werden alle 64 Spielfelder ausgeschnitten, damit das trainierte CNN jedes Feld einzeln klassifizieren kann (siehe Abbildung 23 und Abbildung 24). Mit dem zuvor entzerrten Bild kann dieses Auseinanderschneiden ohne große Probleme durchgeführt werden. So kann das Bild anhand der Eckpunkte in acht gleich große Linien und Reihen eingeteilt werden, wodurch die Grenzen der jeweiligen Felder entstehen. Zu diesen Grenzen wird noch ein Rand an allen Seiten hinzugefügt, um etwaigen Grenzfällen wie kleinen Ungenauigkeiten der Eckenerkennung oder Figuren, die die Spielfeldgrenzen übertreten, zu relativieren.

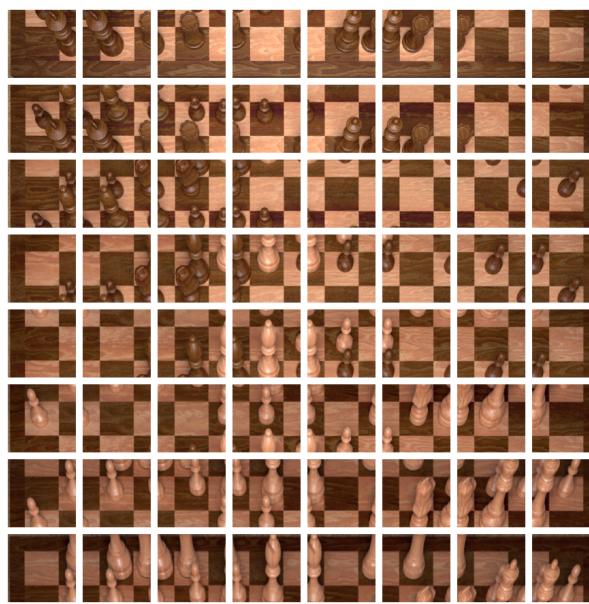


Abbildung 23: Die 64 ausgeschnittenen Schachfelder

Für eine akkurate Klassifizierung der CNNs haben sich unterschiedliche Parameter weitestgehend als unwichtig herausgestellt. Unterschiede zwischen CNNs mit jeweils 1-3 *Convolutional*, *Pooling* und *Fully Connected Layers* waren minimal und jedes getestete CNN konnte auf eine Genauigkeit von über 99% trainiert werden. Als weitaus wichtiger haben

```
array([False,  True,  True, False,  True,  True, False,  True,
       True,  True,  True, False, False, False, False,  True,
       True, False, False, False,  True, False, False,  True,
       True,  True, False,  True, False, False, False, False,
       True, False, False,  True, False, False,  True, False,
       True,  True, False,  True,  True,  True, False, False,
       True,  True, False,  True,  True, False, False,  True,
       False])
```

Abbildung 24: Fertig klassifizierte Occupancy

sich allerdings die Eingabebilder für die CNNs herausgestellt. Hier wurde, wie oben beschrieben, die Umgebung des jeweils zu klassifizierenden Feldes mit betrachtet, was eine Verbesserung von bis zu 0,5% in der Genauigkeit mit sich brachte (verglichen zu trainierten CNNs, welche nur das eigentliche Feld als Eingabebild bekamen). Final wurde für die *Occupancy Classifikation* ein CNN genutzt, welches aus 3 *Convolutional, Pooling* und *Fully Connected Layers* bestand und als Input ein 100x100 Pixel großes Bild bekamen. Von diesen 100x100 Pixeln, nahm das zu klassifizierende Feld 50x50 Pixel ein.

3.3 Umsetzung der Piece Classification

Um die *Piece Classification* umzusetzen, wurde ein weiteres CNN trainiert, um die einzelnen Figuren des Schachspiels und der jeweiligen Positionen erkennen zu können. Der Code für die einzelnen Schritte der *Piece Classification* Pipeline ist in Abbildung 25 abgebildet. Es ist zu erkennen, dass auch hier die Aufbereitung des originalen Inputbildes ähnlich wie bei der *Occupancy Classification* durchgeführt wird.

```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:

    # occupied squares from occupancy classification
    occupied_squares = np.array(self._squares)[occupancy]

    # warp image to regular grid
    warped = create_piece_dataset.warp_chessboard_image(img, corners)

    # cut out squares with pieces on them + further margin around them
    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Imga.fromarray, piece_imgs)
    piece_imgs = map(self._pieces_transforms, piece_imgs)
    piece_imgs = list(piece_imgs)
    piece_imgs = torch.stack(piece_imgs)
    piece_imgs = device(piece_imgs)

    # classify piece images
    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]

    # combine classified pieces with the empty squares
    all_pieces = np.full(len(self._squares), None, dtype=object)
    all_pieces[occupancy] = pieces

    return all_pieces
```

Abbildung 25: Code der Piece Classification

Zunächst wird das Ergebnis der Occupancy Classification genutzt, um ein Array aller belegten Felder zu erhalten. Dieses Array beschreibt die Felder des Schachbrettes Reihe nach Reihe von links nach rechts, beginnend mit der vordersten Reihe (siehe Abbildung 26).

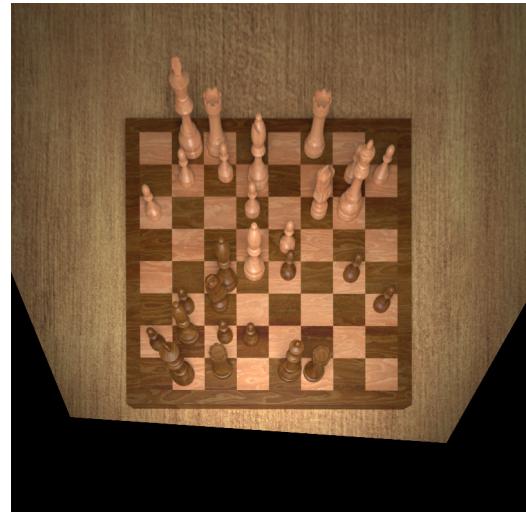
```
array([ 1,  2,  4,  5,  8,  9, 10, 11, 17, 18,
       23, 26, 27, 28, 30, 36, 40, 43, 45,
       46, 49, 50, 51, 54, 55, 57, 58, 61])
```

Abbildung 26: Array der belegten Spielfelder

Danach wird die originale Aufnahme entzerrt, allerdings wird hier ein größerer Rand gelassen, um keine Spielfiguren abzuschneiden (siehe Abbildung 27).



(a) Unbearbeitetes Eingabebild



(b) Entzerrtes Bild des Schachbretts

Abbildung 27: Entzerrung des Schachbrettes

Mit diesem entzerrten Bild und dem oben beschriebenen Array können die belegten Felder ausgeschnitten werden (siehe Abbildung 28). Dieser Prozess wird auch ähnlich wie bei der *Occupancy Classification* durchgeführt, allerdings werden die Felder hier nach unten links ausgerichtet und es wird ein adaptiver Rand gelassen, sodass keine Spielfigur abgeschnitten wird. Falls eine Spielfigur in dem entzerrten Bild nach links zeigt, also außerhalb der linken Grenzen des Feldes tritt, wird dieses Bild gespiegelt (siehe z.B. den König in der ersten Reihe auf dem zweiten Feld in Abb. 27b). So kann garantiert werden, dass jedes Feld an der gleichen Stelle für die Klassifizierung liegt und alle Figuren in dieselbe Richtung zeigen.

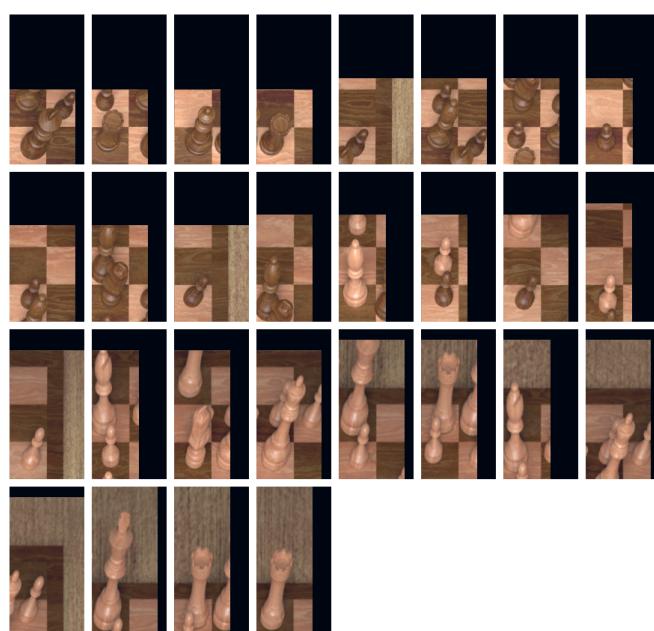


Abbildung 28: Ausgeschnittene Bilder der belegten Spielfelder

Mit den ausgeschnittenen einzelnen Felder wurde ein neu trainiertes CNN genutzt, um die jeweiligen Figuren auf dem Bild zu identifizieren. Anschließend wurden die zuvor durch den *Occupancy Classifier* gefundenen leeren Felder mit den identifizierten Feldern kombiniert und zurückgegeben, um die vollständige Spielposition ausgeben zu können (siehe Abbildung 29).

```
array([Piece.from_symbol('k'), Piece.from_symbol('r'),
       Piece.from_symbol('q'), Piece.from_symbol('r'),
       Piece.from_symbol('p'), Piece.from_symbol('b'),
       Piece.from_symbol('p'), Piece.from_symbol('p'),
       Piece.from_symbol('p'), Piece.from_symbol('n'),
       Piece.from_symbol('p'), Piece.from_symbol('b'),
       Piece.from_symbol('B'), Piece.from_symbol('p'),
       Piece.from_symbol('p'), Piece.from_symbol('P'),
       Piece.from_symbol('P'), Piece.from_symbol('P'),
       Piece.from_symbol('N'), Piece.from_symbol('Q'),
       Piece.from_symbol('P'), Piece.from_symbol('P'),
       Piece.from_symbol('B'), Piece.from_symbol('P'),
       Piece.from_symbol('P'), Piece.from_symbol('K'),
       Piece.from_symbol('R'), Piece.from_symbol('R')], dtype=object)
```

```

. K R . . R . .
. P P B . . P P
P . . P . N Q . .
. . . . P . . .
. . b B p . p .
. p n . . . . p
p b p p . . . .
. k r . q r . .

```

(a) Fertig klassifizierte Occupancy

(b) Fertig klassifizierte Occupancy

Abbildung 29: Prozess der Piece Classification

Beim Training des CNN wurden erneut verschiedene Parameter getestet, um den Einfluss auf die Genauigkeit der Erkennung zu untersuchen. Wie auch bei der *Occupancy Classification* wurde deutlich, dass die Aufbereitung und Standardisierung der Eingabebilder einen weitaus größeren Einfluss auf die erzielte Genauigkeit hatten.

4 Fazit

Mithilfe der Seminararbeit und des Projektes konnte ein tieferes Verständnis für die Thematik der digitalen Aufzeichnung von Schachspielen über Bilder geschaffen werden. Im Projekt wurden dabei verschiedene Ansätze erprobt, um die einzelnen Schritte der *Board Localisation*, *Occupancy Classification* und *Piece Classification* umzusetzen. Auf Grundlage des behandelten Papers und des im Zuge dessen entwickelten Codes gelang es eine zusammenhängende Anwendung mit der Verwendung eines Datensatzes und des Trainings von CNNs zu schaffen.

Deutlich wurde hierbei die Schwierigkeit, die einzelnen Schritte der Schachspiel-Erkennung mit den Open-CV eigenen Funktionen umzusetzen. Besonders die Berichtigung von Fehlern in Grenzfällen stellte sich als schwierig heraus. Aufgrund der Komplexität des Themas und bereits bestehender Ansätze viel es hierbei schwer einen vollständig eigenen Ansatz zu entwickeln, was das Thema zukünftiger Forschungen darstellen könnte.

Eine Erweiterung der aktuellen Klassifizierungspipeline könnte eine automatische Klassifizierung eines gesamten Schachmatches darstellen. So könnte als Input nicht nur ein Bild einer einzelnen Schachposition eingegeben werden, sondern ein Videofeed oder eine Bilderreihe von jedem Spielzug. So könnte der Klassifizierer ein gesamtes Schachspiel für eine spätere Analyse digitalisieren. Des Weiteren muss aktuell die Farbe angegeben werden, aus welcher Ansicht das Eingabebild aufgenommen wurde. Dies ist wichtig für eine korrekte Unterscheidung der Figurfarben, welche je nach Belichtung (z.B. mit hellen Spotlights) sonst falsch erkannt wurde. Mit weiteren Optimierungen könnte dieser benötigte Input vermutlich auch entfernt werden.

Abbildungsverzeichnis

1	Visualisierung des Cluster-Algorithmus	3
2	Finden vertikaler und horizontaler Linien mittels DBSCAN	3
3	Eliminieren gleicher Linien	4
4	Occupancy Classification (siehe Wölflein u. Arandjelović, 2021c, S.7) . . .	5
5	Piece Classification (siehe Wölflein u. Arandjelović, 2021c, S.9)	6
6	Anwendung der <code>findChessboardCorners()</code> -Methode	7
7	Anwendung der <code>drawChessboardCorners()</code> -Methode	7
8	Anwendung der <code>goodFeaturesToTrack()</code> -Methode	8
9	Ausgabe vor Entzerrung des Bildes	9
10	Ausgabe nach Entzerrung des Bildes	9
11	Anwendung der <code>findHomography()</code> -Methode	10
12	Anwendung des Canny-Edge-Detectors und der Hough Transformation . .	10
13	Ausgabe vor Warping des Bildes nach Hough Transformation	11
14	Ausgabe nach Warping des Bildes nach Hough Transformation	11
15	Definierte Config-Datei	12
16	Definition der <code>detectEdges()</code> -Methode	12
17	Definition der <code>detectLines()</code> -Methode	13
18	RANSAC-Schleife	13
19	Berechnung der Homografie und der finalen Eckpunkte	14
20	Ausgabe der fertigen Board Localisation	14
21	Code der Occupancy Classification	15
22	Entzerrung des Schachbrettes	16
23	Die 64 ausgeschnittenen Schachfelder	16
24	Fertig klassifizierte Occupancy	17
25	Code der Piece Classification	18
26	Array der belegten Spielfelder	18
27	Entzerrung des Schachbrettes	19
28	Ausgeschnittene Bilder der belegten Spielfelder	19
29	Prozess der Piece Classification	20

Literaturverzeichnis

[OpenCV-Dokumentation 2022a] OPENCV-DOKUMENTATION: *Camera Calibration and 3D Reconstruction.* https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html.

Version: 2022

[OpenCV-Dokumentation 2022b] OPENCV-DOKUMENTATION: *Canny()*.
https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de. Version: 2022

[OpenCV-Dokumentation 2022c] OPENCV-DOKUMENTATION: *drawChessboardCorners()*.
https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga6a10b0bb120c4907e5eabbcd22319022. Version: 2022

[OpenCV-Dokumentation 2022d] OPENCV-DOKUMENTATION: *Feature Detection*.
https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html. Version: 2022

[OpenCV-Dokumentation 2022e] OPENCV-DOKUMENTATION: *findChessboardCorners()*.
https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga93efa9b0aa890de240ca32b11253dd4a. Version: 2022

[OpenCV-Dokumentation 2022f] OPENCV-DOKUMENTATION: *findHomography()*.
https://docs.opencv.org/3.4/d1/de0/tutorial_py_feature_homography.html. Version: 2022

[OpenCV-Dokumentation 2022g] OPENCV-DOKUMENTATION: *goodFeaturesToTrack()*.
https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga1d6bb77486c8f92d79c8793ad995d541. Version: 2022

[OpenCV-Dokumentation 2022h] OPENCV-DOKUMENTATION: *HoughLines()*.
https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga46b4e588934f6c8dfd509cc6e0e4545a. Version: 2022

[Wölflein u. Arandjelović 2021a] WÖLFLEIN, Georg ; ARANDJELOVIĆ, Ognjen: *chesscog*.
<https://github.com/georg-wolflein/chesscog>. Version: 2021

[Wölflein u. Arandjelović 2021b] WÖLFLEIN, Georg ; ARANDJELOVIĆ, Ognjen: *Dataset of Rendered Chess Game State Images*. <https://osf.io/xf3ka/>. Version: 2021

[Wölflein u. Arandjelović 2021c] WÖLFLEIN, Georg ; ARANDJELOVIĆ, Ognjen: Determining chess game state from an image. In: *Journal of Imaging* 7 (2021), Nr. 6, S. 94