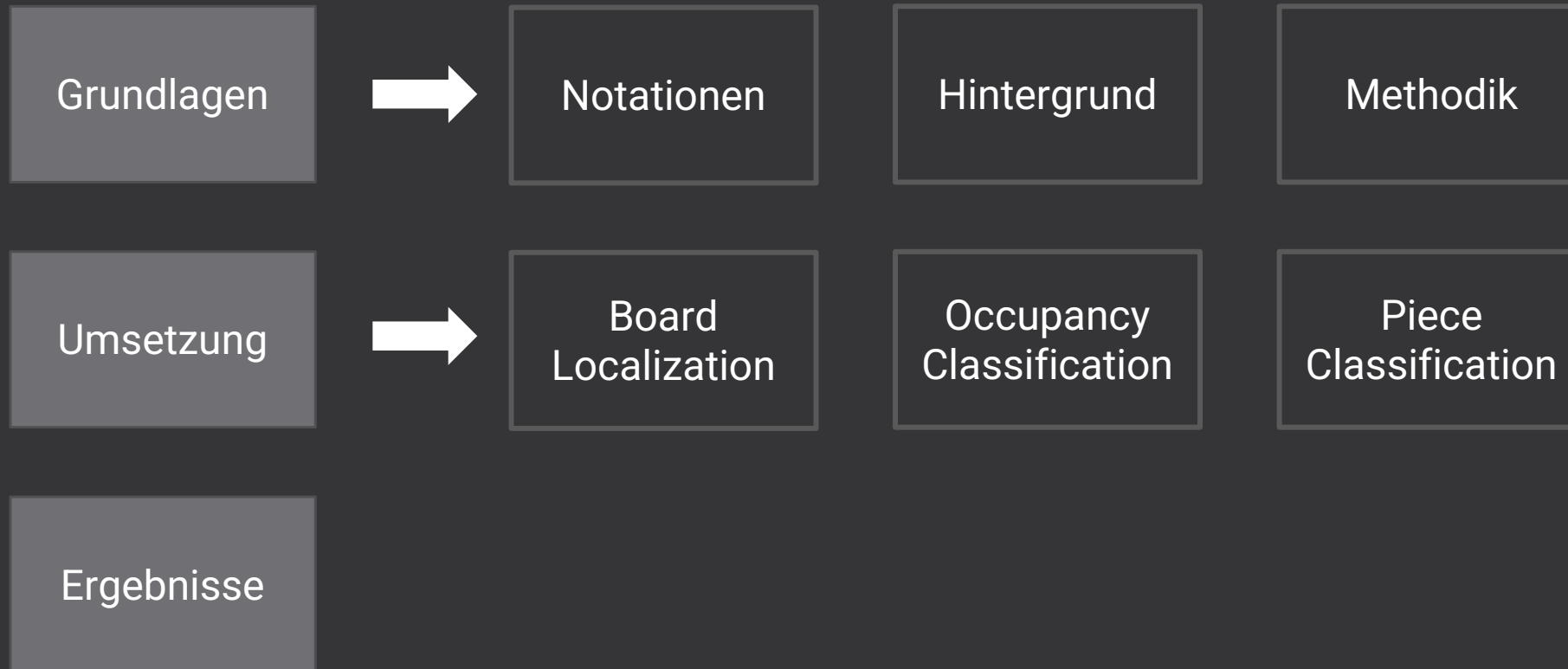


Bildbasierte digitale Aufzeichnung realer Schachspiele

Aileen Jurkosek und Julian Hardtung

Inhalt



Grundlagen



Algebraic Chess Notation

- King: **K**
- Queen: **Q**
- Rook: **R**
- Bishop: **B**
- Knight: **N**
- Pawn: keine Abkürzung

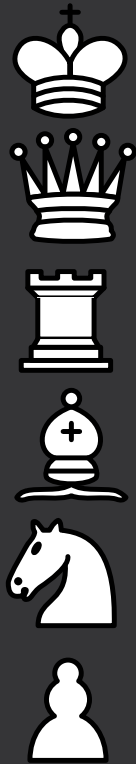
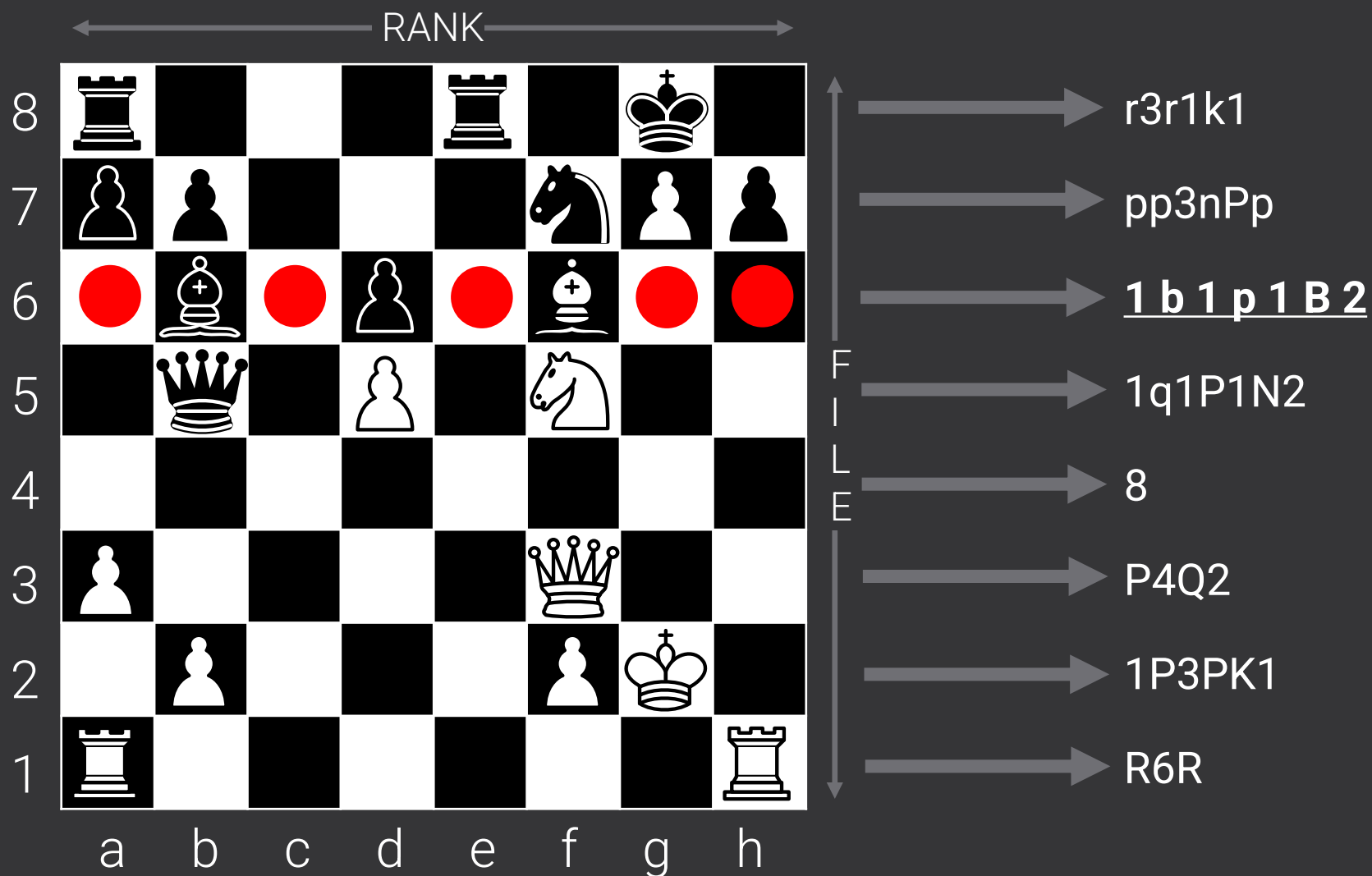


Diagram illustrating the Rank and File notation on an 8x8 chessboard. The horizontal axis is labeled **RANK** (1 to 8) and the vertical axis is labeled **FILE** (a to h). The board shows alternating black and white squares, with the bottom-left square (a1) being black.

	a	b	c	d	e	f	g	h
8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1

Forsyth–Edwards Notation (FEN)



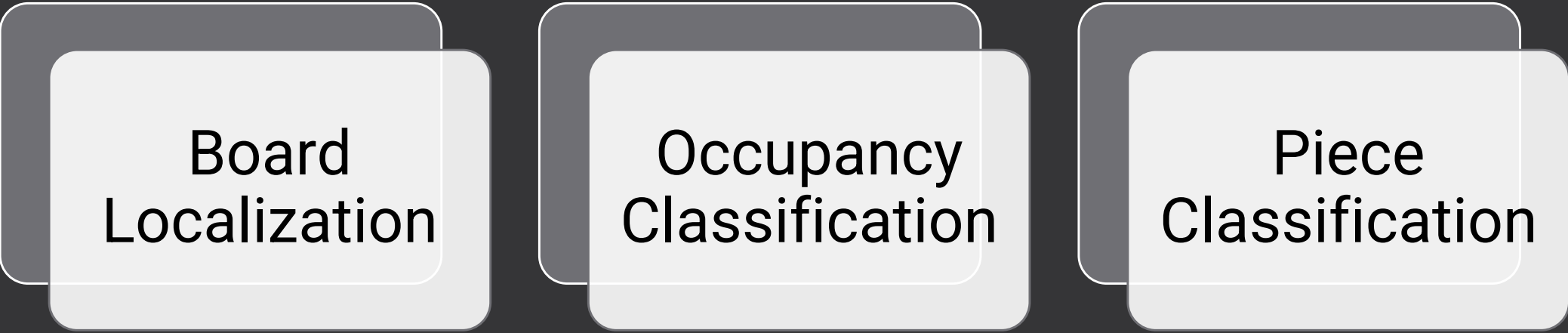
Hintergrund

- Durchführung eines physischen Schachspiels
- Festhalten **einzelner Spielpositionen** mit Fotos
- Eingabe der Positionen an einem Computer

→ Möglichkeit zur Analyse des Spiels

→ **Automatisierung** der Eingabe weniger aufwändig oder fehleranfällig

Methodik



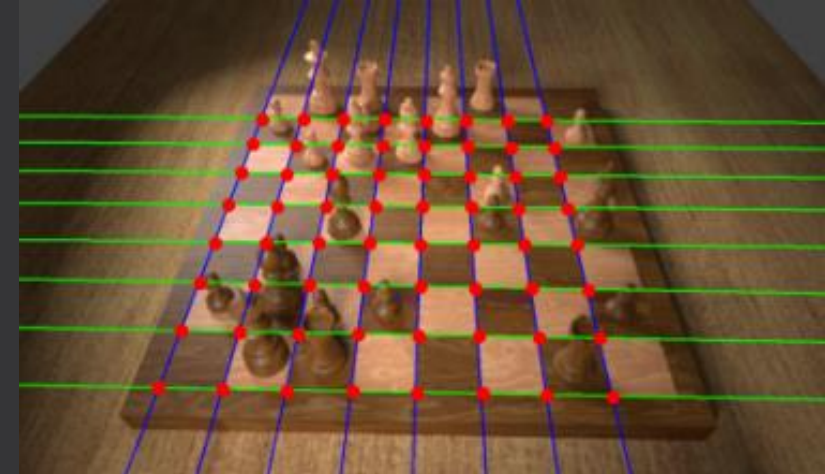
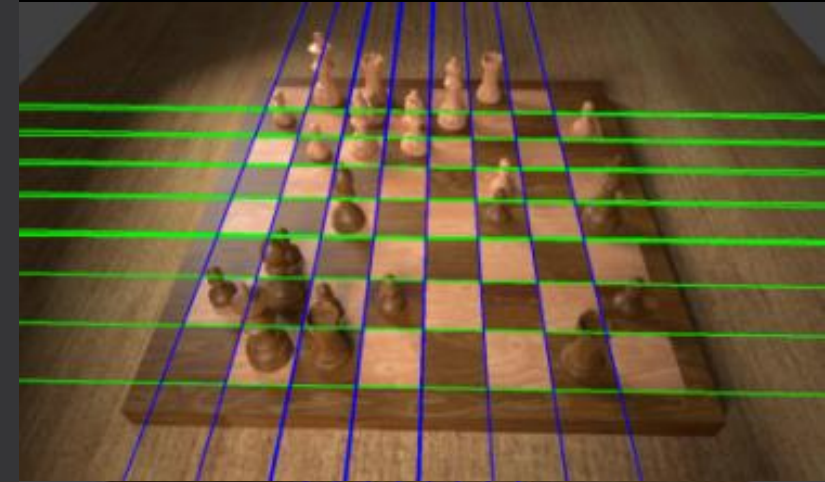
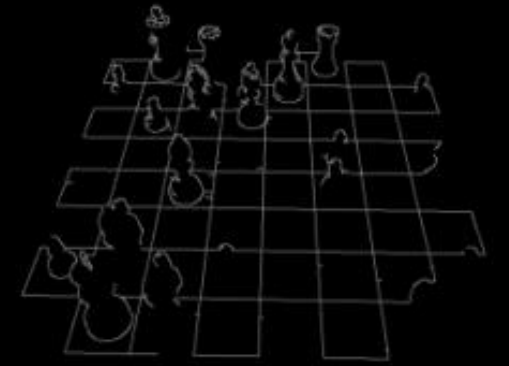
Board
Localization

Occupancy
Classification

Piece
Classification

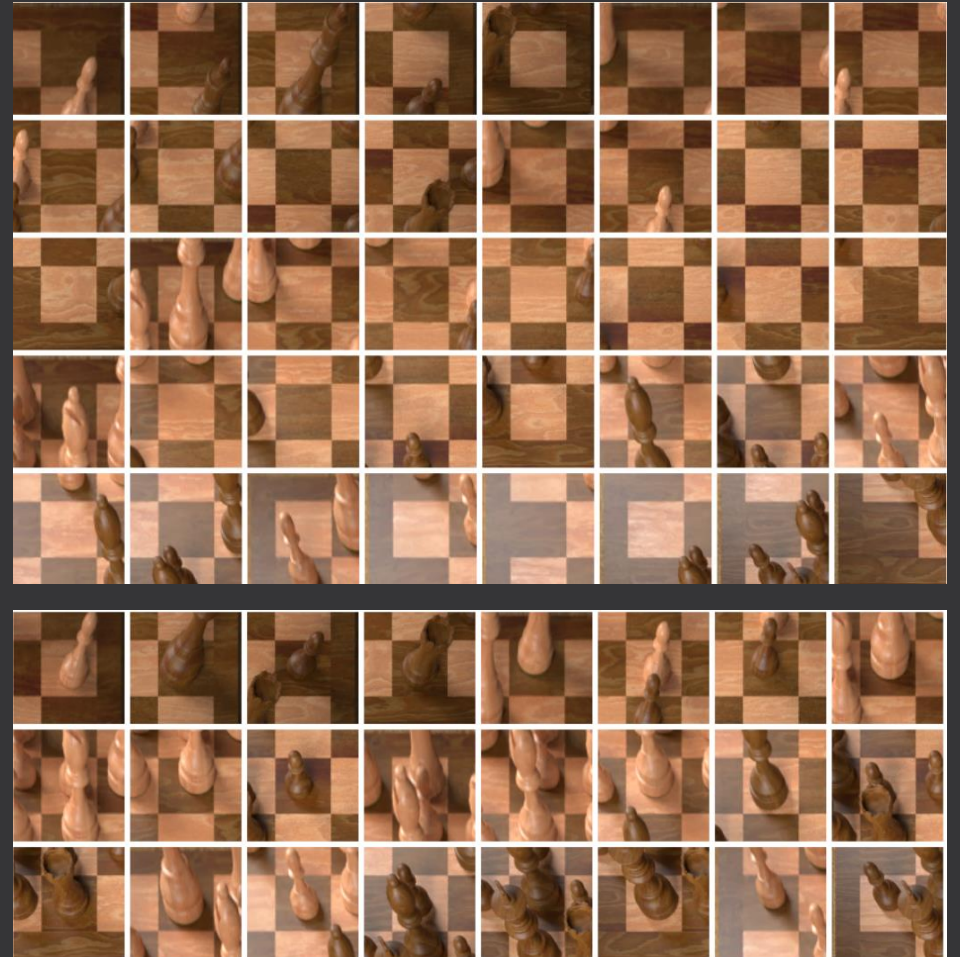
Board Localization - Theorie

- Schachbrett soll digital erkannt werden
- Finden der Schnittpunkte
 - Anwendung auf Grauwertbild
 - Durchführung möglich mit Canny Edge Detector, Hough Transformation und Clustern der gefundenen Punkte
- Berechnung der Homografie
 - Warping des Eingabebildes
 - Durchführung einer projektiven Transformation



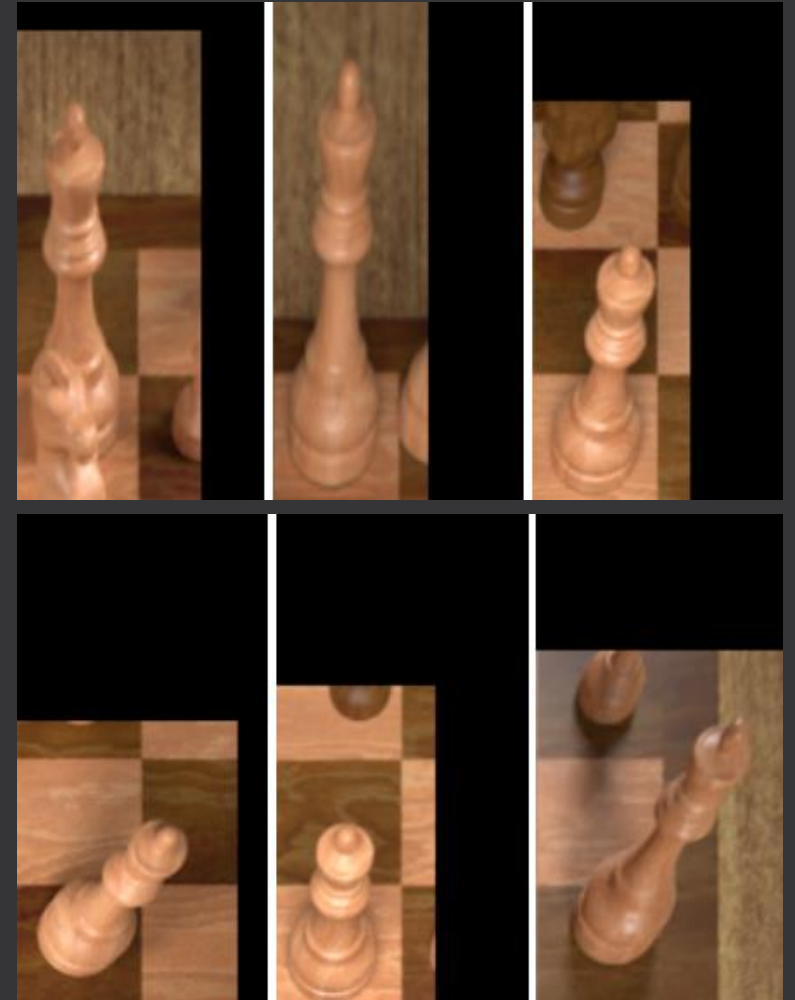
Occupancy Classification - Theorie

- Erkennung, ob Felder des Schachbrettes belegt oder frei sind
- Training von CNNs mit verschiedenen Input Bildern und verschiedenen Parametern
- Modelle liefern Output Units welche die Klassen Leer und Belegt repräsentieren



Piece Classification - Theorie

- Erkennung der verschiedenen Figuren
 - Bauer, Läufer, Springer, Turm, Dame, König
 - Je weiß oder schwarz
- Bildausschnitt wird als Eingabe gegeben
- Ausgabe liefert Schachfigur auf dem entsprechenden Feld
- Erweiterung der Bounding Boxes zur Vermeidung abgeschnittener Figuren



Umsetzung



1. Board Localization

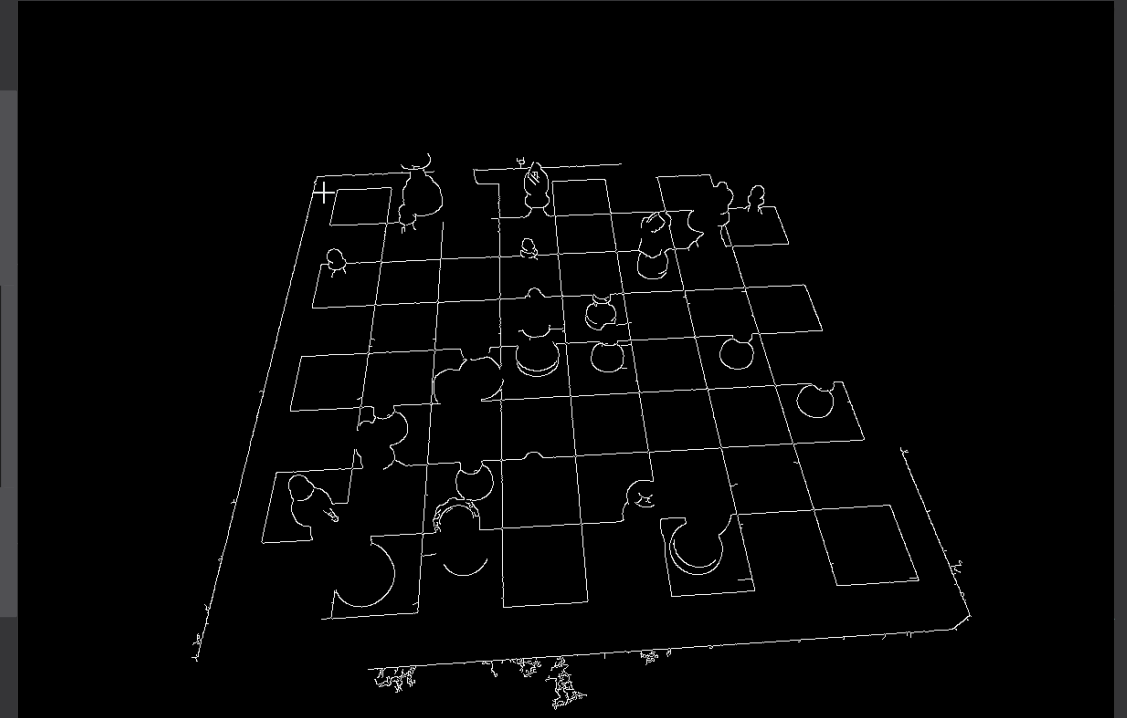
```
def find_corners(img: np.ndarray):  
    img, img_scale = resize_img(img)  
  
    grey_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)  
    edges_img = detect_edges(grey_img)  
    lines = detect_lines(edges_img)  
  
    if lines.shape[0] > 400:  
        raise Exception("too many lines found")  
  
    all_horizontal_lines, all_vertical_lines = cluster_horizontal_and_vertical_lines(lines)  
  
    horizontal_lines = eliminate_similar_lines(all_horizontal_lines, all_vertical_lines)  
    vertical_lines = eliminate_similar_lines(all_vertical_lines, all_horizontal_lines)  
  
    all_intersection_points = get_intersection_points(horizontal_lines, vertical_lines)
```

- Konvertierung in Grauwertbild
- Finden der Ecken und Linien im Bild



1. Board Localization

```
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
```

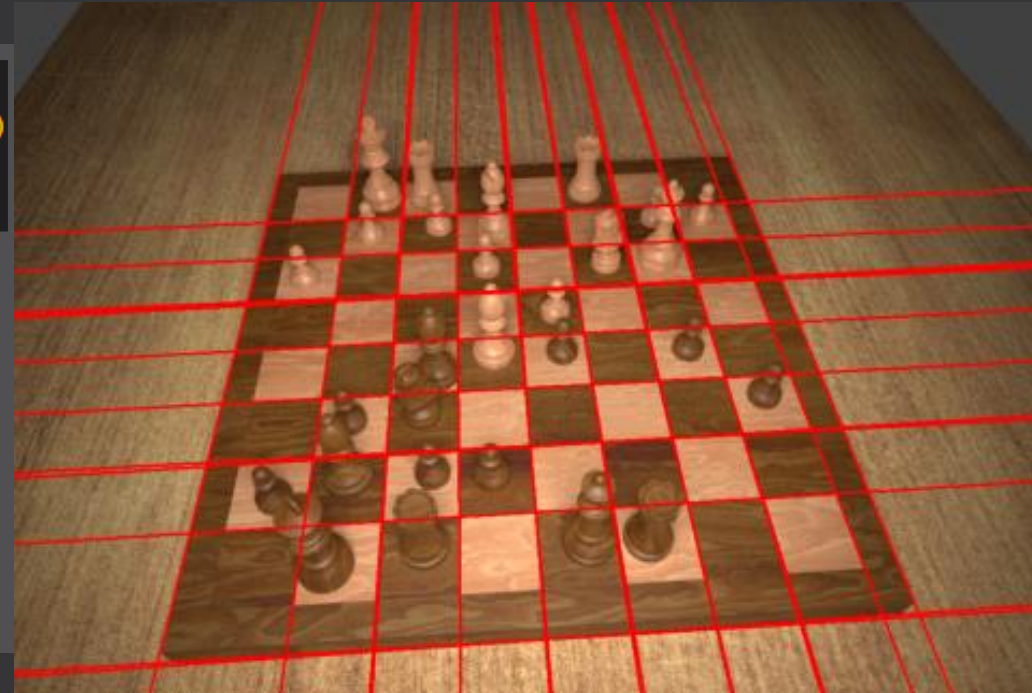


- Ecken finden mit Canny-Edge-Detector

1. Board Localization

```
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
```



- Linien finden mit Hough-Transformation

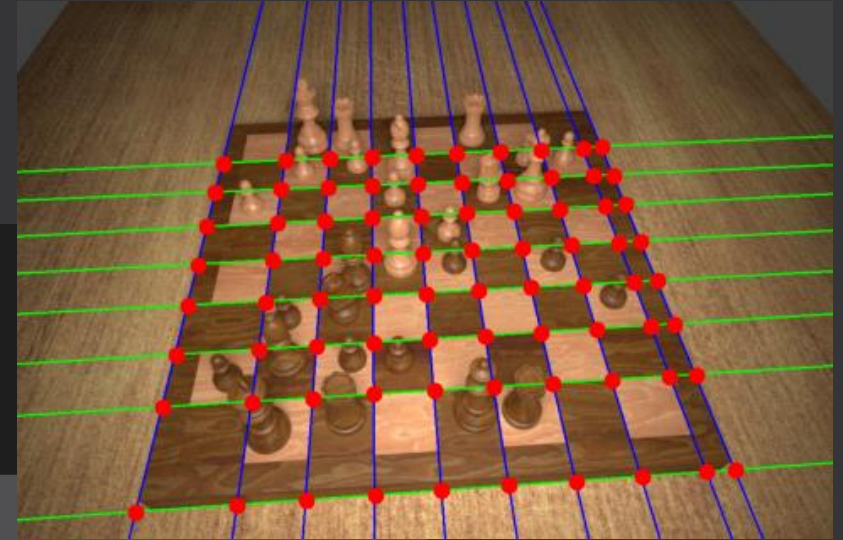
1. Board Localization

```
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")
```



- Clustern der horizontalen und vertikalen Linien
- Berechnung der Schnittpunkte
- Berechnung der Homografie-Matrix über RANSAC-Algorithmus

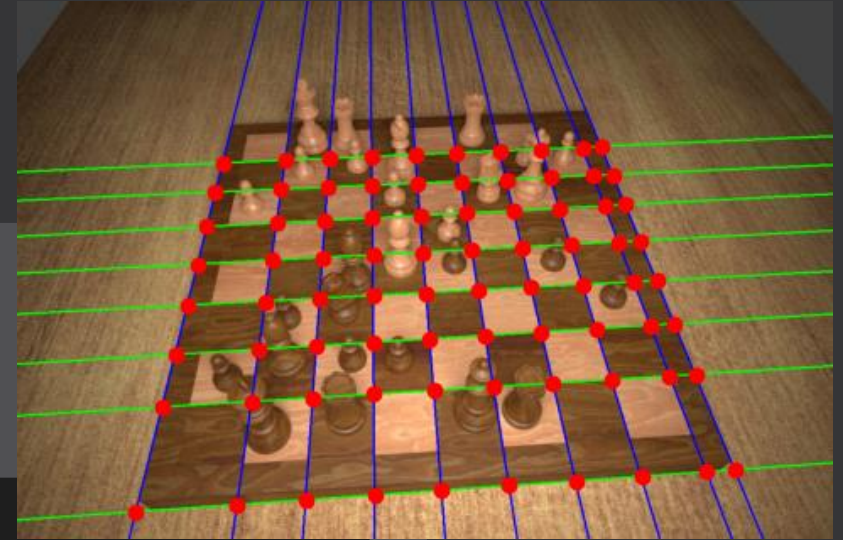
1. Board Localization

```
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")
```



- Berechnung der Homografie-Matrix über RANSAC-Algorithmus
- Wiederholung, bis festgelegter Schwellenwert erreicht ist

1. Board Localization

```
# 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
```

- Berechnung der Homografie
- Perspektive des Bildes/der Koordinaten korrigieren



1. Board Localization

```
# 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)
```



- Vertikale und horizontale Grenzen finden
- 4 Eckpunkte des Schachbretts ausgeben

2. Occupancy Classification

```
def _classify_occupancy(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying occupancy of the chessboard")

    warped = warp_chessboard_image(img, corners)
    square_imgs = map(functools.partial(crop_square, warped, turn=turn), self._squares)
    square_imgs = map(Image.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)
    occupancy = self._occupancy_model(square_imgs)
    occupancy = occupancy.argmax(axis=-1) == self._occupancy_cfg.DATASET.CLASSES.index("occupied")
    occupancy = occupancy.cpu().numpy()

    return occupancy
```



- Verzerren des Bildes basierend auf den gefundenen Eckpunkten des Schachbrettes

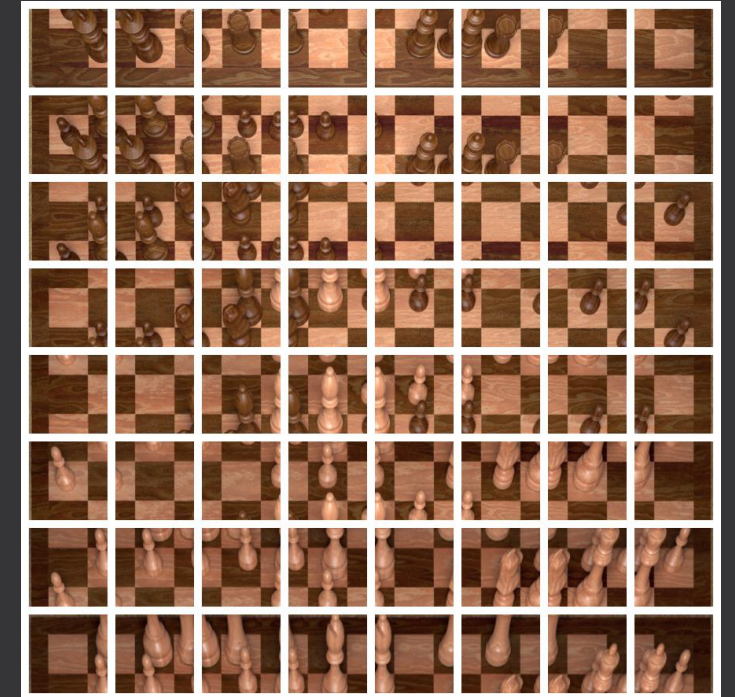
2. Occupancy Classification

```
def _classify_occupancy(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying occupancy of the chessboard")

    warped = warp_chessboard_image(img, corners)
    square_imgs = map(functools.partial(crop_square, warped, turn=turn), self._squares)
    square_imgs = map(Image.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)

    occupancy = self._occupancy_model(square_imgs)
    occupancy = occupancy.argmax(axis=-1) == self._occupancy_cfg.DATASET.CLASSES.index("occupied")
    occupancy = occupancy.cpu().numpy()

    return occupancy
```



- Ausschneiden der einzelnen Felder des entzerrten Schachbrettes
 - Mit zusätzlichem Rand

2. Occupancy Classification

```
def _classify_occupancy(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying occupancy of the chessboard")

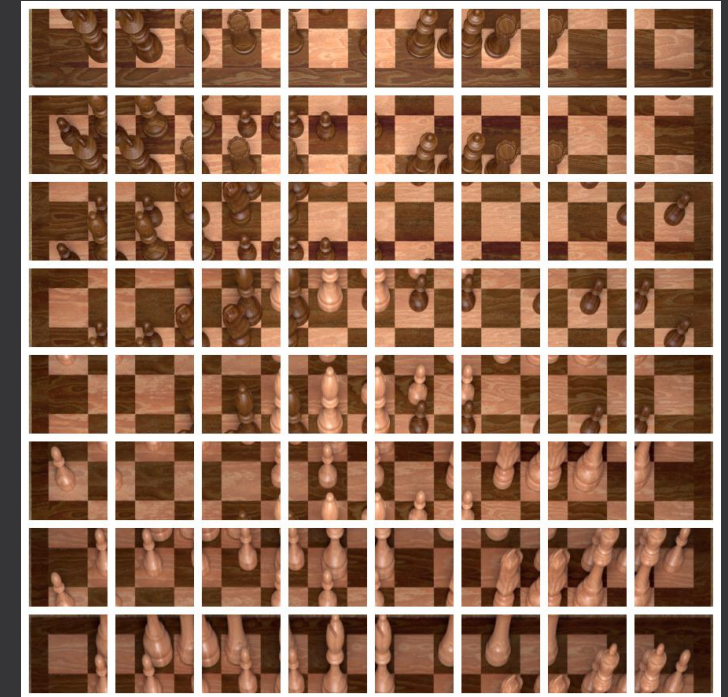
    warped = warp_chessboard_image(img, corners)
    square_imgs = map(functools.partial(crop_square, warped, turn=turn), self._squares)
    square_imgs = map(Image.fromarray, square_imgs)
    square_imgs = map(self._occupancy_transforms, square_imgs)
    square_imgs = list(square_imgs)
    square_imgs = torch.stack(square_imgs)
    square_imgs = device(square_imgs)

    occupancy = self._occupancy_model(square_imgs)
    occupancy = occupancy.argmax(axis=-1) == self._occupancy_cfg.DATASET.CLASSES.index("occupied")
    occupancy = occupancy.cpu().numpy()

    return occupancy
```

- Klassifizieren

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



3. Piece Classification

```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying pieces on occupied spaces of the chessboard")

    occupied_squares = np.array(self._squares)[occupancy]
    warped = create_piece_dataset.warp_chessboard_image(img, corners)
    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Image.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)
    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]
    all_pieces = np.full(len(self._squares), None, dtype=np.object)
    all_pieces[occupancy] = pieces

    return all_pieces
```

```
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])
```

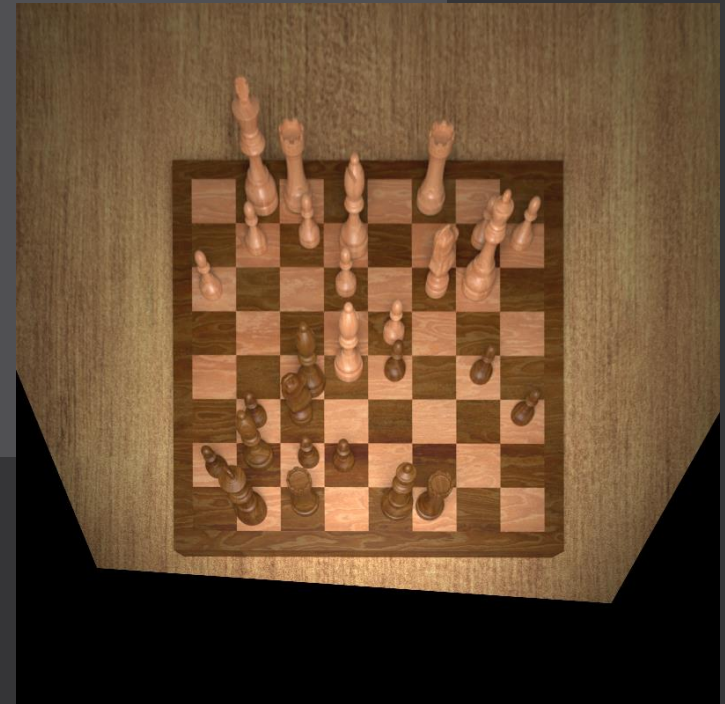


3. Piece Classification

```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying pieces on occupied spaces of the chessboard")

    occupied_squares = np.array(self._squares)[occupancy]
    warped = create_piece_dataset.warp_chessboard_image(img, corners)
    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Image.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)
    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]
    all_pieces = np.full(len(self._squares), None, dtype=np.object)
    all_pieces[occupancy] = pieces

    return all_pieces
```



- Entzerre die einzelnen Bilder der Figuren

3. Piece Classification

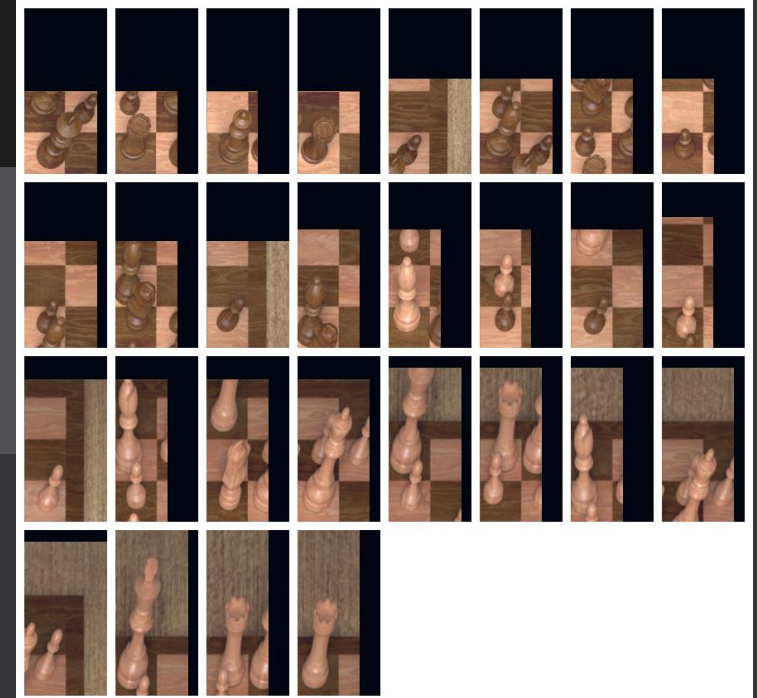
```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying pieces on occupied spaces of the chessboard")

    occupied_squares = np.array(self._squares)[occupancy]
    warped = create_piece_dataset.warp_chessboard_image(img, corners)

    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Image.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)

    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]
    all_pieces = np.full(len(self._squares), None, dtype=np.object)
    all_pieces[occupancy] = pieces

    return all_pieces
```



- Ausschneiden der einzelnen Felder des entzerrten Schachbrettes
 - Mit zusätzlichem Rand

3. Piece Classification

```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying pieces on occupied spaces of the chessboard")

    occupied_squares = np.array(self._squares)[occupancy]
    warped = create_piece_dataset.warp_chessboard_image(img, corners)
    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Image.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)
    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]
    all_pieces = np.full(len(self._squares), None, dtype=np.object)
    all_pieces[occupancy] = pieces

    return all_pieces
```

```
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)
```

- Identifiziere die Spielfiguren mit dem trainierten CNN

3. Piece Classification

```
def _classify_pieces(self, img: np.ndarray, turn: chess.Color, corners: np.ndarray, occupancy: np.ndarray) -> np.ndarray:
    print("_____")
    print("Classifying pieces on occupied spaces of the chessboard")

    occupied_squares = np.array(self._squares)[occupancy]
    warped = create_piece_dataset.warp_chessboard_image(img, corners)
    piece_imgs = map(functools.partial(create_piece_dataset.crop_square, warped, turn=turn), occupied_squares)
    piece_imgs = map(Image.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)
    pieces = self._pieces_model(piece_imgs)
    pieces = pieces.argmax(axis=-1).cpu().numpy()
    pieces = self._piece_classes[pieces]

    all_pieces = np.full(len(self._squares), None, dtype=np.object)
    all_pieces[occupancy] = pieces

    return all_pieces
```

- Kombiniere die leeren Felder mit den vorher identifizierten Feldern und gib sie zurück

.	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	.	.

Ergebnisse

Corner Detection: 0.179 s
Occupancy Classification: 0.1019 s
Piece Classification: 0.0753 s
Preparing Results: 0.0011 s

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

You can view this position at <https://lichess.org/editor/1KR2R2/1PPB2PP/P2P1NQ1/4P3/2pBp1p1/1pn4p/pbpp4/1kr1qr2>



<https://lichess.org/editor/1KR2R2/1PPB2PP/P2P1NQ1/4P3/2pBp1p1/1pn4p/pbpp4/1kr1qr2>

Aussicht

Aktuelle Probleme / Limitierungen:

- Jeder Spielzug muss bisher **einzel**n fotografiert und durch das System identifiziert werden
- Bei jeder Identifikation muss die **Farbe angegeben** werden, aus welcher Sicht das Bild aufgenommen wurde
- Jede identifizierte Schachposition steht für sich und kann aktuell nicht (**automatisch**) zu einem kompletten Schachspiel **kombiniert** werden