AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Chess game registration

Vision Systems in Robotics

Authors:	Jan Gallina Jakub Górski
Field of study:	Automatic Control and Robotics
Specialization:	Komputerowe Systemy Sterowania
Semester:	I

Cracow, 2025 r.

Table of contents

1. Introduction

1.1. The Objective

The goal of this project was to record a game of chess using a webcam. In order to achieve that, we decided to build an algorithm that would monitor each tile of the chessboard separately and use a convolutional neural network (CNN) to detect and classify chess pieces in real time.

1.2. The Overview of Used Technologies

During this project, we made use of various technologies from different areas. To accomplish the project goals, the following tools and libraries were used:

- Python language – main programming language used for implementation,
- OpenCV library – for image processing and computer vision tasks,
- TensorFlow library – to build, train and run the convolutional neural network (CNN),
- NumPy library – for numerical operations and matrix manipulation,
- scikit-learn library – to evaluate the results of the learning phase of the CNN.

From the hardware point of view, we must have:

- a PC/laptop,
- a webcam,
- a chessboard,
- 2D images of chess pieces.

Figure 1 *– the research site*

2. Method Overview

The following sections present the steps taken throughout the project. At the beginning of each subsection, a brief description of a program component is provided, followed by the corresponding code and the results obtained.

2.1. Perspective Transformation

Let us start with presenting the defined constants, variables at the beginning of the Python file (figure 2) and the user-created functions (figure 3). They will be used in the following parts of the task.

```python
if __name__ == "__main__":
    # constants
    chessboard_columns_names = 'abcdefgh'
    chessboard_rows_names = '12345678'
    straight_lines_intersection_pattern = np.zeros((3, 3, 3), dtype=np.uint8)
    straight_lines_intersection_pattern[:, :, 2] = np.array([[0, 255, 0], [255, 255, 255], [0, 255, 0]], dtype=np.uint8)
    intersection_region_side_len = 10
    chess_pieces_cnn_model = load_model('model_cnn.keras')
    chess_pieces_names = ["CG", "CK", "CS", "CP", "CH", "CW", "XX", "BG", "BK", "BS", "BP", "BH", "BW"]
    chess_pieces_scan_photos_path = './tiles_scan/photos'

    # variables
    persp_change_input_coords = []
    straight_lines_intersection_coords_matrix = np.zeros(shape=(9, 9), dtype=tuple)
    chessboard = [["XX" for _ in range(8)] for _ in range(8)]
```

Figure 2 *– the defined constants and variables*

```python
# fuctions
def click_event(event, x, y, flags, params): # in order to get the coordinates of the chessboard from the input camera image
    if event == cv2.EVENT_LBUTTONDOWN:       # LBUTTONDOWN - click the left mouse button
        persp_change_input_coords.append((x, y))

def convert_to_float(image, label):          # convert image from uint8 to float32
    image = convert_image_dtype(image, dtype=tf.float32)
    return image, label
```

Figure 3 *– the user-created functions*

The next goal was to set the connection with the camera. The proper definition is presented in the figure 4.

```python
# camera options
cap = cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 800)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 800)
```

Figure 4 *– connecting to the camera*

The initial attempts to detect the chessboard structure using edge detection algorithms revealed, that the process was highly sensitive to the appearance of different objects in the camera FOV. This problem is also related to the manual tuning of the parameters used for edge detection, which was performed.

To solve this issue, a perspective transformation was applied. At the start of the program, the user is asked to manually select the four corners of the chessboard using the mouse, based on the live webcam feed. These points are then used to calculate a transformation matrix, which adjusts the image so that the chessboard is properly aligned and centered. The procedure is called the 2D homography (figures 5 and 6).

```
# getting coords for the perspetive change - 2D homography
cv2.namedWindow("Select chessboard")
cv2.setMouseCallback("Select chessboard", click_event)
while len(persp_change_input_coords) < 4:
    _, frame = cap.read()
    cv2.imshow("Select chessboard", frame) # the first coords - the top left corner, the second coords - the top right corner
    cv2.waitKey(1)
#cv2.imwrite("./photos/chessboard_view_from_camera.jpg", frame)
cv2.destroyWindow("Select chessboard")
```

Figure 5 *– selecting corners of the chessboard*

After selecting corners by the user, the perspective change matrix is calculated. Then, it is used to align chessboard in the center of the image. The input and the output for the algorithm are presented, respectfully, in the figure 7 and the figure 8.

```
# the perspective change matrix M
top_view_first_width = int(np.sqrt(((persp_change_input_coords[1][0]-persp_change_input_coords[0][0])**2)+
                                   ((persp_change_input_coords[1][1]-persp_change_input_coords[0][1])**2)))
top_view_second_width = int(np.sqrt(((persp_change_input_coords[2][0]-persp_change_input_coords[3][0])**2)+
                                    ((persp_change_input_coords[2][1]-persp_change_input_coords[3][1])**2)))
top_view_width = max(top_view_first_width, top_view_second_width)

top_view_first_height = int(np.sqrt(((persp_change_input_coords[0][0]-persp_change_input_coords[3][0])**2)+
                                    ((persp_change_input_coords[0][1]-persp_change_input_coords[3][1])**2)))
top_view_second_height = int(np.sqrt(((persp_change_input_coords[1][0]-persp_change_input_coords[2][0])**2)+
                                     ((persp_change_input_coords[1][1]-persp_change_input_coords[2][1])**2)))
top_view_height = max(top_view_first_height, top_view_second_height)

persp_change_input_coords = np.float32(persp_change_input_coords)
persp_change_top_view_coords = np.float32([[0, 0], [top_view_width-1, 0], [top_view_width-1,
                                                    top_view_height-1], [0, top_view_height-1]])

M = cv2.getPerspectiveTransform(persp_change_input_coords, persp_change_top_view_coords)

# naming the chessboard tiles (for the tile name - view from camera dictionary)
tiles_names = [f"{col}{row}" for row in chessboard_rows_names for col in chessboard_columns_names]

# finding tiles
_, frame = cap.read()

# the perspective change - 2D homography
top_view = cv2.warpPerspective(frame, M, (top_view_width, top_view_height), flags=cv2.INTER_LINEAR)
```

Figure 6 *– calculating the perspective change matrix and warping the perspective*

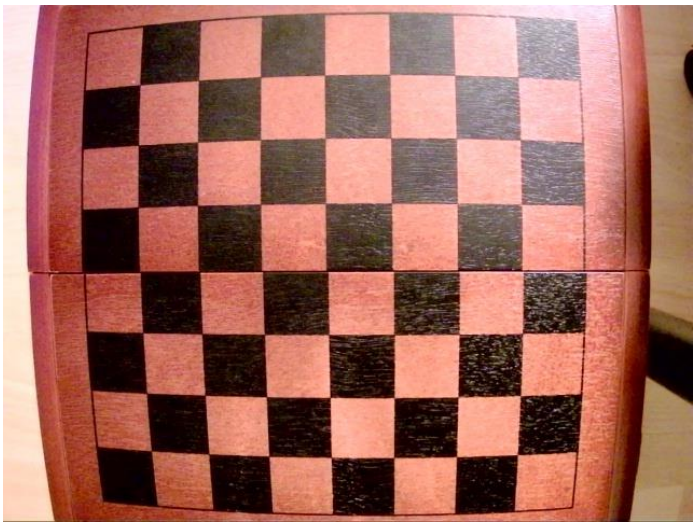

Figure 7 *– the original image from the camera*

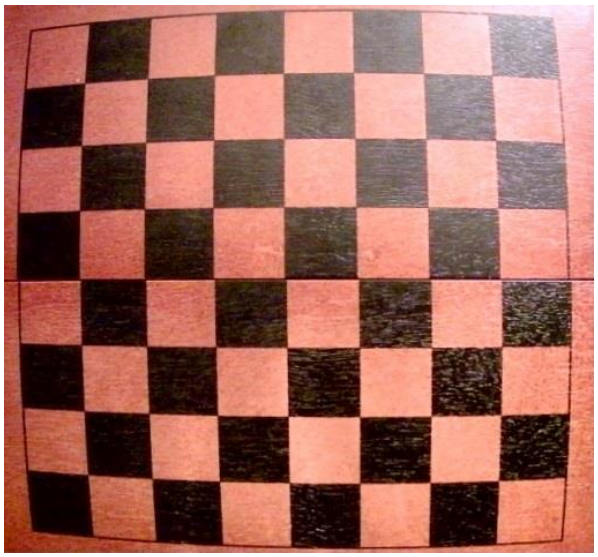

Figure 8 *– the image after warping the perspective*

After these operations, the image contains only the chessboard, with all other elements removed. It is centered and perfectly aligned with the camera view, which makes the edge detection algorithm more reliable.

2.2. Edge Detection

After applying the perspective transformation, the next step was to detect the edges of the chessboard using the Canny Edge Detection algorithm, introduced during lectures. Based on the transformed image of the empty chessboard (without pieces), the algorithm identifies the edges. Then, using the Hough Transform, the detected edges are drawn as straight lines on the image. This approach makes it possible to extract all individual squares of the chessboard. However, without the earlier perspective correction, this method was too sensitive to variations in the camera view.

```
# light equalization - CLAHE algorithm
lab = cv2.cvtColor(top_view, cv2.COLOR_BGR2LAB)
l, a, b = cv2.split(lab)
clahe = cv2.createCLAHE(clipLimit=4.0, tileGridSize=(8, 8))
clahe_l = clahe.apply(l)
clahe_lab = cv2.merge((clahe_l, a, b))
clahe_top_view = cv2.cvtColor(clahe_lab, cv2.COLOR_LAB2BGR)
```

Figure 9 *– the CLAHE algorithm*

The CLAHE (Contrast Limited Adaptive Histogram Equalization) algorithm was used to enhance the visibility of details in both overexposed and underexposed areas of the image captured by the webcam. This step helped to improve the overall contrast of the chessboard image.

The procedure is based on computing the histogram equalization on the smaller parts of the image, but the pixels that exceeds the clip limit are uniformly distributed among the rest of the bins of the histogram.

The comparison between the images before and after performing CLAHE are shown in the figure 10.

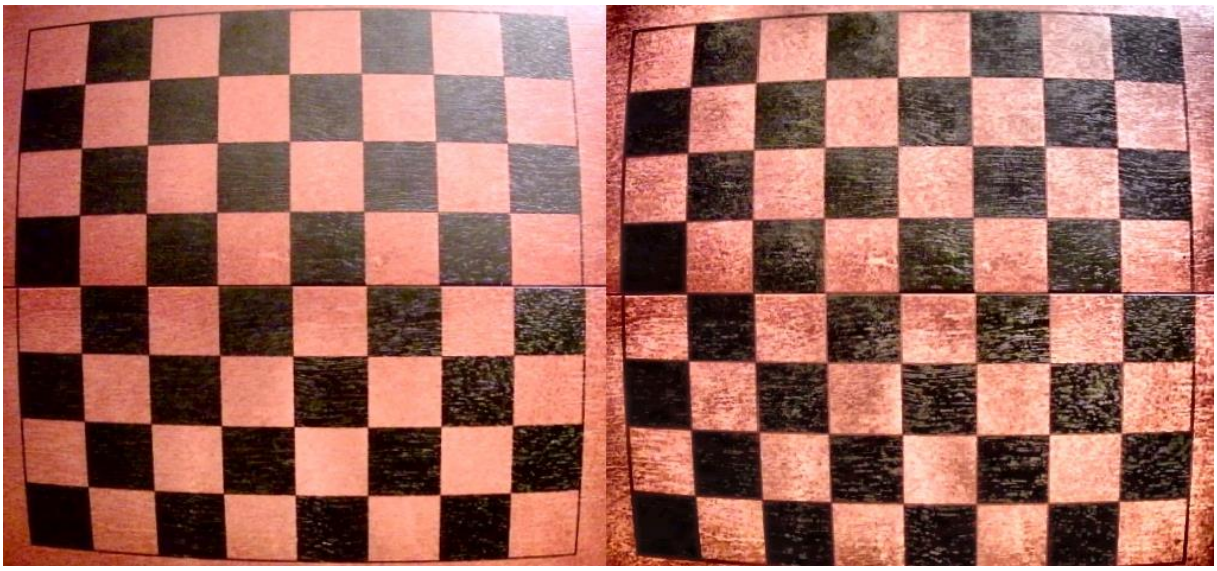

Figure 10 *– before and after the CLAHE algorithm*

The image on the right is more consistent in lighting and it doesn't have overexposed and underexposed areas.

```
# smoothing
gaussian = cv2.GaussianBlur(clahe_top_view, ksize=(7, 7), sigmaX=3.0)
#cv2.imwrite("./photos/smoothing.jpg", gaussian)

# grayscale
gray = cv2.cvtColor(gaussian, cv2.COLOR_BGR2GRAY)

# Canny edge detection
edges = cv2.Canny(gray, threshold1=60.0, threshold2=150.0)
#cv2.imwrite("./photos/canny.jpg", edges)
```

Figure 11 *– smoothing and Canny Edge Detection*

Then, we performed the Canny Edge Detection method. The result is a binary image called edges, which contains only the detected edges.

Figure 12 *– the image after smoothing*

The image from the figure 12 was then used to detect edges using Canny.

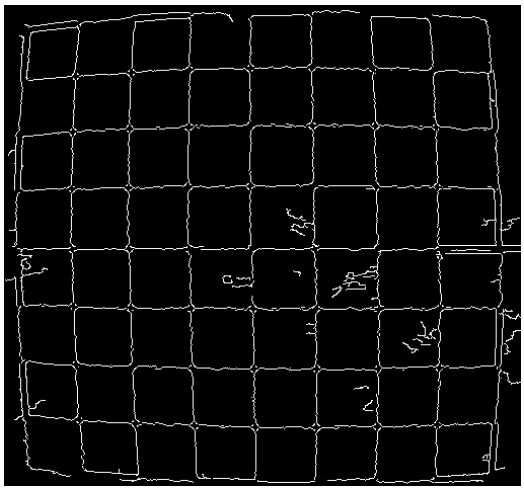

Figure 13 *– the result of Canny Edge Detection*

Next, the Hough Transform was used to detect vertical and horizontal lines, which are necessary for the process of extracting individual chessboard squares (figure 14). The lines were drawn on the original image from the camera (figure 15).

```
# finding horizontal and vertical lines - Hough transform
lines = cv2.HoughLines(edges, rho=1, theta=np.pi/180, threshold=50)
top_view_with_hor_and_ver_lines = copy.deepcopy(top_view) # in order to draw lines on the top_view image and not change the top_view image itself
lines_view = np.zeros(np.shape(top_view), dtype=np.uint8)
if lines is not None:
    for r_and_theta in lines:
        r, theta = r_and_theta[0]
        if np.isclose(theta, 0, atol=0.01) or np.isclose(theta, np.pi, atol=0.01): # finding vertical lines
            x1, x2 = int(r), int(r)
            y1, y2 = 0, top_view_height-1
        elif np.isclose(theta, np.pi/2, atol=0.01): # finding horizontal lines
            x1, x2 = 0, top_view_width-1
            a = -np.cos(theta)/np.sin(theta)
            b = r/np.sin(theta)
            y1 = int(a*x1+b)
            y2 = int(a*x2+b)
        else:
            continue
        # limiting lines
        y1 = max(0, min(top_view_height-1, y1))
        y2 = max(0, min(top_view_height-1, y2))
        cv2.line(lines_view, (x1, y1), (x2, y2), (0, 0, 255), 1) # drawing red lines - thickness: 1
        cv2.line(top_view_with_hor_and_ver_lines, (x1, y1), (x2, y2), (0, 0, 255), 1)
#cv2.imwrite("./photos/horizontal_and_vertical_lines_on_chessboard.jpg", top_view_with_hor_and_ver_lines)
```

Figure 14 *– the Hough transform*

As you can see in the code. We only seek lines that are approximately horizontal or vertical.

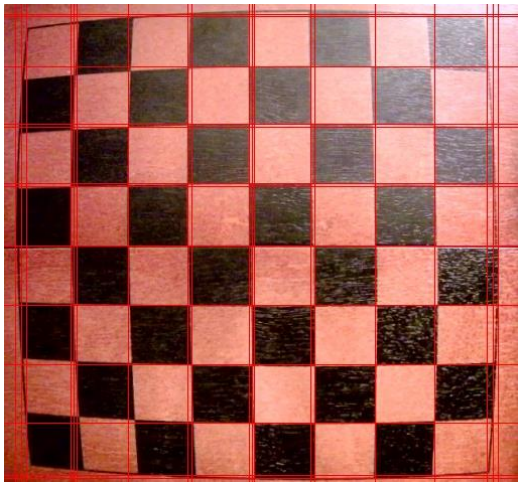

Figure 15 *– the image from camera with vertical and horizontal lines*

2.3. Chessboard Square Extraction

Using the lines detected with the Hough Transform, the image was divided into 64 individual squares. The first step in extracting the squares was to find the intersection points of the detected lines. Then, a dictionary was created that mapped each square's name (e.g., C1, B8) to its image part. These square regions were later used by the neural network to detect chess pieces.

```
# finding intersections of the lines (finding tiles)
intersection_view = np.zeros(np.shape(top_view), dtype=np.uint8)
intersection_row = 0
intersection_col = 0
for i in range(1, top_view_height-1):
    for j in range(1, top_view_width-1):
        window = lines_view[i-1:i+2, j-1:j+2, :]
        if np.all(window == straight_lines_intersection_pattern): # finding intesection
            if np.any(intersection_view[max(i-intersection_region_side_len, 0):min(i+intersection_region_side_len+1, top_view_height),
                                        max(j-intersection_region_side_len, 0):min(j+intersection_region_side_len+1, top_view_width), 2]
                                        == 255): # there are no intersection in the neighbourhood of the potential new intersection
                continue
            else:
                intersection_view[i, j, 2] = 255
                straight_lines_intersection_coords_matrix[intersection_row, intersection_col] = (i, j)
                intersection_col += 1
                if intersection_col > 8:
                    intersection_row += 1
                    intersection_col = 0
                if intersection_row > 8:
                    break
    if intersection_row > 8:
        break
```

Figure 16 *– finding intersections*

This part of the code (figure 16) identifies the intersection points of the detected vertical and horizontal lines, which correspond to the corners of the chessboard squares. These points are stored in a matrix and later used to extract individual tiles from the board. To identify the intersection, we used the *red plus* pattern. Also, due to multiple horizontal/vertical lines being drawn close to each other, we had to set the radius for each detected intersection, so that new detection wouldn't be found near the already marked corners.

```
# naming the chessboard tiles (for the tile name - view from camera dictionary)
tiles_names = [f"{col}{row}" for row in chessboard_rows_names for col in chessboard_columns_names]
```

Figure 17 *– naming the tiles*

```
while True:
    _, frame = cap.read()

    # the perspective change - 2D homography
    top_view = cv2.warpPerspective(frame, M, (top_view_width, top_view_height), flags=cv2.INTER_LINEAR)

    # viewing
    cv2.imshow("View", top_view)

    # the dictionary of the tiles view
    tiles_view = []
    for i in range(8):
        for j in range(8):
            y_first, x_first = straight_lines_intersection_coords_matrix[8-i-1, j]
            _, x_last= straight_lines_intersection_coords_matrix[8-i-1, j+1]
            y_last, _ = straight_lines_intersection_coords_matrix[8-i, j]
            tile = np.array(top_view[y_first:y_last, x_first:x_last, :], dtype=np.uint8)
            tiles_view.append(tile)
    tiles = {names:views for (names, views) in zip(tiles_names, tiles_view)}
```

Figure 18 *– extracting tiles from the image*

The above part of the program stores each square's name and its view in a dictionary. In the following steps, these images are used to identify which chess piece, if any, is present on each square.

2.4. Piece Detection

The final step of the program is detecting whether a chess piece is present on a given square. Iterating through all the squares identified in the previous step, the convolutional neural network predicts the probability of each square containing a specific piece. Based on this, the program creates an interpretation of the board state, which is printed to the console. After each move, the user must press the “a” key to trigger a new scan of the board. The current board layout is then displayed in the console.

```
        key = cv2.waitKey(1)
        if key == 97: # 'a'
            #cv2.imwrite("./photos/chessboard_pattern_.jpg", top_view)
            # chess game registration
            tile_order = iter(tiles)
            # scanning all the tiles to predict which chess piece is on it
            for i in range(8*8):
                # cleaning the folder where the tile scan will be stored
                for photo in os.listdir(chess_pieces_scan_photos_path):
                    tile_scan_path = os.path.join(chess_pieces_scan_photos_path, photo)
                    if os.path.isfile(tile_scan_path):
                        os.remove(tile_scan_path)
                tile_name = next(tile_order)
                # scanning
                cv2.imwrite(chess_pieces_scan_photos_path + "/scan.jpg", tiles[tile_name])
                # load scan
                scan = image_dataset_from_directory(
                    directory='./tiles_scan',
                    image_size=[224, 224], # resizing the images to the given shape
                    interpolation='nearest', # the interpolation type while resizing
                    batch_size=1, # the number of images loaded at the same time
                    shuffle=False # randomize
                )
                scan = (
                    scan
                    .map(convert_to_float) # mapping (convert from uint8 to float)
                    .cache() # use cache to speed up iterations
                    .prefetch(buffer_size=AUTOTUNE) # prefetch the new batch while the earlier one is still being processed
                )
```

Figure 19 *– scanning each tile and preparing for the CNN*

Each tile is scanned and saved to a directory. The image is then resized and converted to float format to be processed by the CNN.

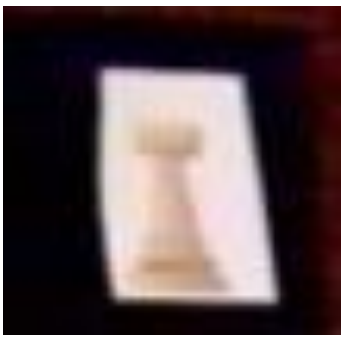

Figure 20 *– the example of the scanned tile*

This is what a scanned tile looks like. Based on this image, the CNN will predict what piece is present on the tile.

```
                tile_predict = chess_pieces_cnn_model.predict(scan) # returns the propabilities that the shown piece belongs to given class
                # finding the class of the shown piece
                figure_idx = np.argmax(tile_predict, axis=1)
                chessboard[7-i//8][i%8] = chess_pieces_names[figure_idx[0]]
            # showing the register game
            for i in range(8):
                print(chessboard[i])
                print("\n")
        if key == 27: # 'Esc'
            break
```

Figure 21 *– using CNN and saving the chessboard configuration in the matrix*

After each move, the user can press the "a" key on the keyboard, and the program will detect the current arrangement of pieces on the chessboard and display its interpretation as the text output in the console. The prepared results are shown in the figures 22-24.

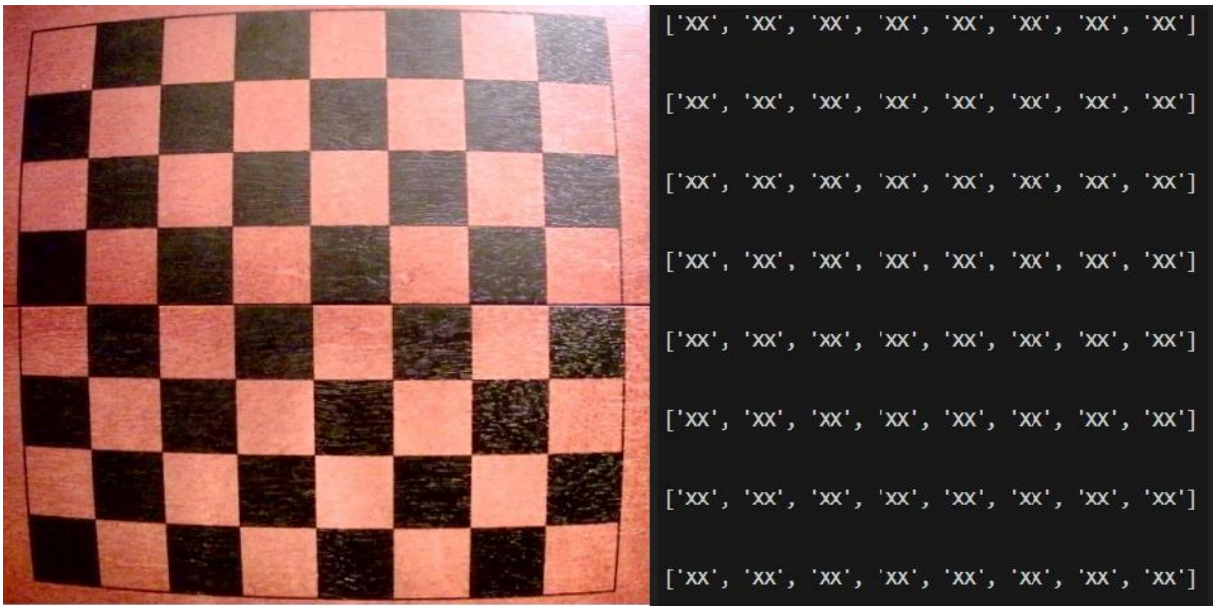

Figure 22 *– the representation of the empty board*

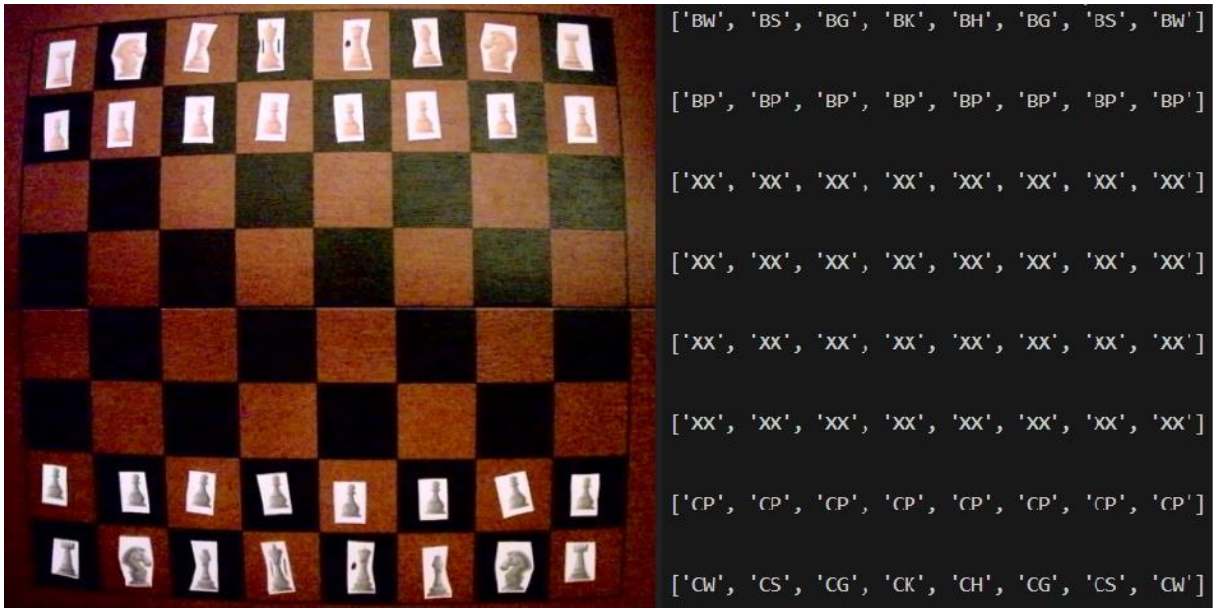

Figure 23 *– the representation of the starting position*

Each piece has its own name. (e.g. BW – *Biała Wieża,* CS – *Czarny Skoczek,* XX – empty tile).

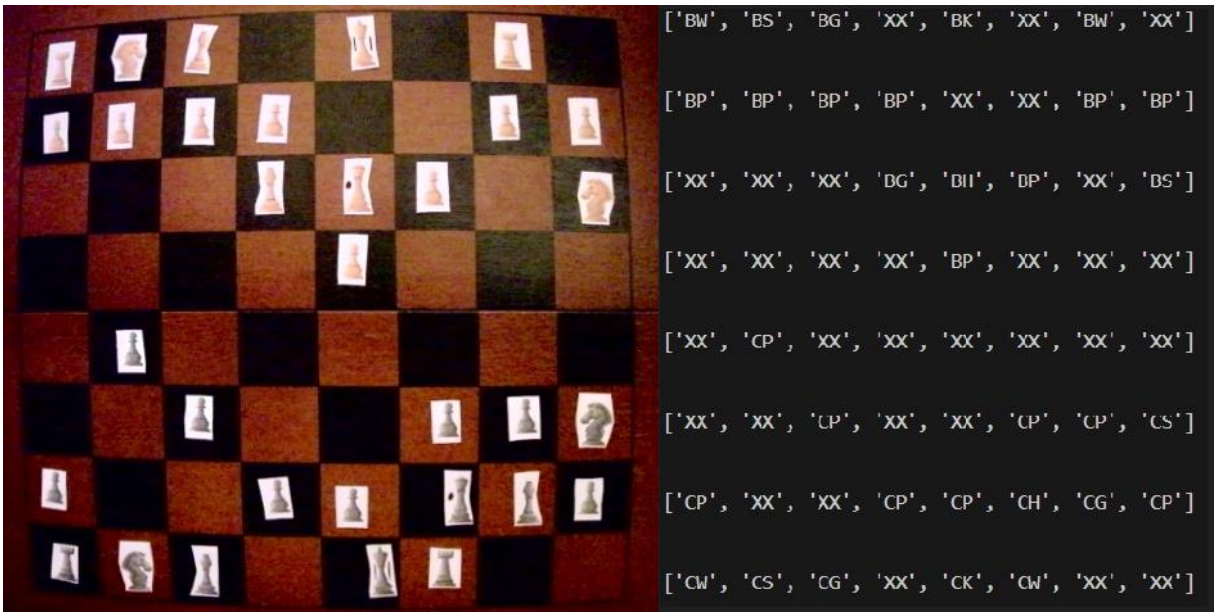

Figure 24 *– the representation of the mid game*

In each scenario, the neural network correctly interpreted the image from the camera of the arranged figures.

We attach the whole program to the report.

3. The CNN Model

This chapter describes the stages of preparing, training, and testing the neural network for recognizing chess pieces on a chessboard. This solution is largely based on the article *"Detecting Chess Pieces with a CNN"* by Thomas S. Visser. The network was created in a separate program and saved to a *.keras* file, that was later used in the main program.

```
# data import (and labelling)
train_images = image_dataset_from_directory(
  directory='./dataset/train',
  labels='inferred', # generated based on the directory structure
  label_mode='categorical', # encoding of labels - in this case: categorical vector
  image_size=[224, 224], # resizing the images to the given shape
  interpolation='nearest', # the interpolation type while resizing
  batch_size=32, # the number of images loaded at the same time
  shuffle=True, # randomize images
)
```

Figure 25 *– loading the data to the notebook (categorical labelling) – train data*

```
# convert the images to float
train_images = (
  train_images
  .map(convert_to_float) # mapping
  .cache() # use cache to speed up iterations
  .prefetch(buffer_size=AUTOTUNE) # prefetch the new batch while the earlier one is still being processed
)
```

Figure 26 *– converting the matrices to float – train data*

The first step after importing the libraries is to load the data into the test, validation, and training sets using a function from the TensorFlow library (figures 25, 26). Data is split in proportion:

- 60% training set,
- 20% validate set,
- 20% test set.

Train dataset

	Category	Count
0	black_bishop	192
1	black_king	192
2	black_knight	192
3	black_pawn	191
4	black_queen	192
5	black_rook	192
6	empty_tile	192
7	white_bishop	192
8	white_king	192
9	white_knight	192
10	white_pawn	190
11	white_queen	192
12	white_rook	192

Test dataset

	Category	Count
0	black_bishop	64
1	black_king	63
2	black_knight	64
3	black_pawn	61
4	black_queen	63
5	black_rook	64
6	empty_tile	64
7	white_bishop	64
8	white_king	64
9	white_knight	64
10	white_pawn	64
11	white_queen	63
12	white_rook	64

Valid dataset

	Category	Count
0	black_bishop	64
1	black_king	64
2	black_knight	64
3	black_pawn	63
4	black_queen	64
5	black_rook	64
6	empty_tile	64
7	white_bishop	64
8	white_king	64
9	white_knight	64
10	white_pawn	64
11	white_queen	64
12	white_rook	64

***Figure 27** – the division of the data between the sets*

Additionally empty space is also classified in those sets as *empty_tile.*

Next, the process described in the article is followed. Transfer learning (using the already trained CNN) is applied by using a pre-trained VGG16 model trained on the large ImageNet dataset. The early convolutional layers are frozen to preserve their learned low-level features, while only the later layers and a new "head" are trained on the smaller chess piece dataset.

The head consists of two hidden layers with 256 nodes each, batch normalization layers to normalize inputs and speed up the training, and a dropout rate of 0.4 to prevent overfitting. Data augmentation techniques like random contrast, horizontal flips, and slight translations are used to increase data diversity.

The model is compiled with the Adam optimizer and the categorical cross-entropy loss, suitable for the 13-class classification task. Early stopping is implemented to halt the training if

validation loss does not improve after 30 epochs, ensuring the model does not overfit and the best weights are restored.

```
pretrained_model = VGG16(weights='imagenet', include_top=False, input_shape=(224, 224, 3)) # use 'imagenet' weights in CNN and delete the fully-connected layers
pretrained_model.summary()
pretrained_model.trainable = False # do not train any layer

# the head of the model (the last layers)
model = tf.keras.Sequential([
    # data augmentation
    RandomContrast(factor=0.5),
    RandomFlip(mode='horizontal'),
    RandomTranslation(height_factor=0.1, width_factor=0.1),
    # base model
    pretrained_model,
    # the head
    # block 1
    layers.BatchNormalization(),
    layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    # block 2
    layers.Conv2D(filters=256, kernel_size=3, activation='relu', padding='same'),
    tf.keras.layers.GlobalMaxPooling2D(),
    layers.Dropout(0.4),
    # output
    layers.BatchNormalization(),
    layers.Dense(13, activation='softmax'),
])

early_stopping = EarlyStopping(
    min_delta = 0.001,          # min improvement
    patience = 30,              # epochs to wait before stopping
    restore_best_weights = True, # the weights corresponding with the lowest validation loss
)

optimizer = tf.keras.optimizers.Adam(epsilon=0.001)

# compile
model.compile(
    optimizer= optimizer,
    loss='categorical_crossentropy',
    metrics=['categorical_accuracy'],
)
```

Figure 28 *– the CNN model*

Next, the network is trained, and the results are displayed on various plots where the improvement of the network's performance can be visually observed from epoch to epoch.

```
with tf.device('/GPU:0'):
    history = model.fit(
        train_images,
        validation_data= valid_images,
        epochs= 200,
        callbacks=[early_stopping],
       )
model.save("model_cnn.keras")
```

```
Epoch 1/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 178s 2s/step - categorical_accuracy: 0.6718 - loss: 1.0954 - val_categorical_accuracy: 0.5740 - val_loss: 1.7009
Epoch 2/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 172s 2s/step - categorical_accuracy: 0.9835 - loss: 0.0763 - val_categorical_accuracy: 0.8303 - val_loss: 1.0861
Epoch 3/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 172s 2s/step - categorical_accuracy: 0.9894 - loss: 0.0532 - val_categorical_accuracy: 0.9615 - val_loss: 0.4089
Epoch 4/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 172s 2s/step - categorical_accuracy: 0.9970 - loss: 0.0223 - val_categorical_accuracy: 0.9868 - val_loss: 0.1738
Epoch 5/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 173s 2s/step - categorical_accuracy: 0.9955 - loss: 0.0215 - val_categorical_accuracy: 0.9892 - val_loss: 0.0686
Epoch 6/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 172s 2s/step - categorical_accuracy: 0.9989 - loss: 0.0114 - val_categorical_accuracy: 0.9916 - val_loss: 0.0539
Epoch 7/200
78/78 ━━━━━━━━━━━━━━━━━━━━ 172s 2s/step - categorical_accuracy: 0.9968 - loss: 0.0156 - val_categorical_accuracy: 0.9747 - val_loss: 0.0954
```

Figure 29 *– training the CNN*

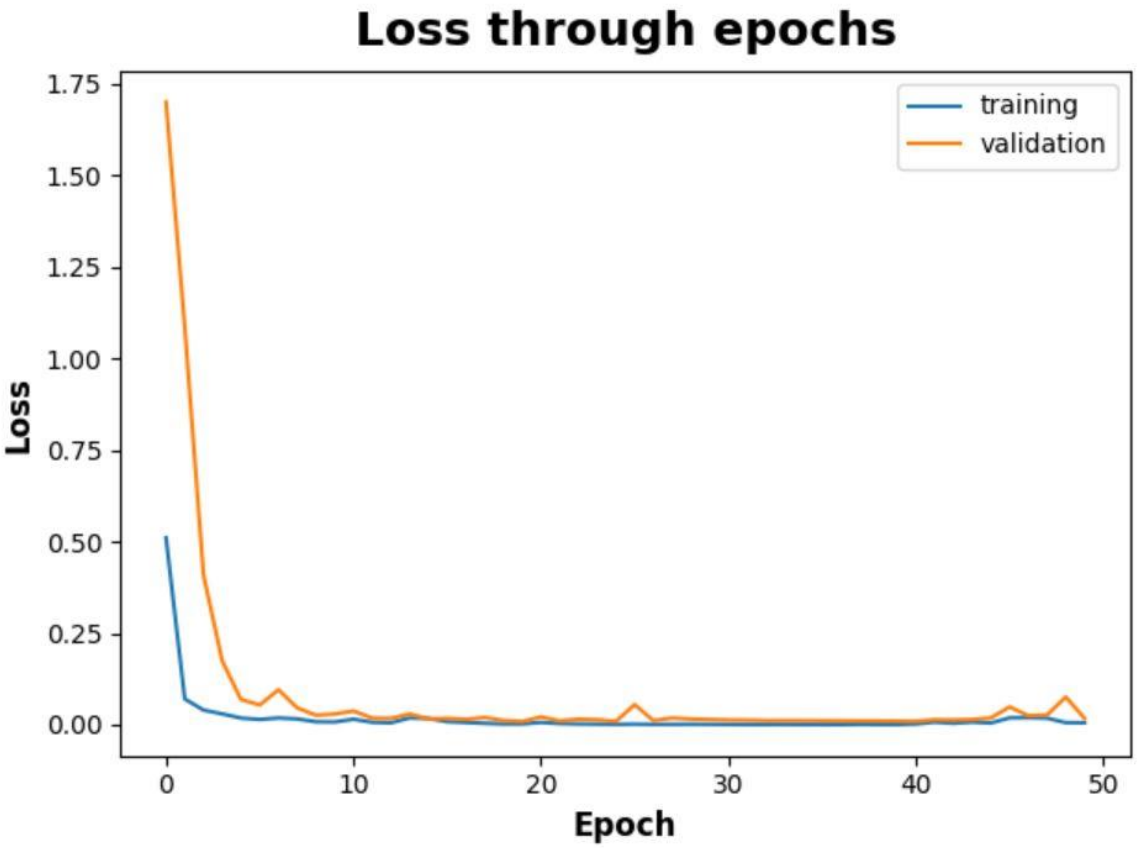

Figure 30 *– loss through epochs*

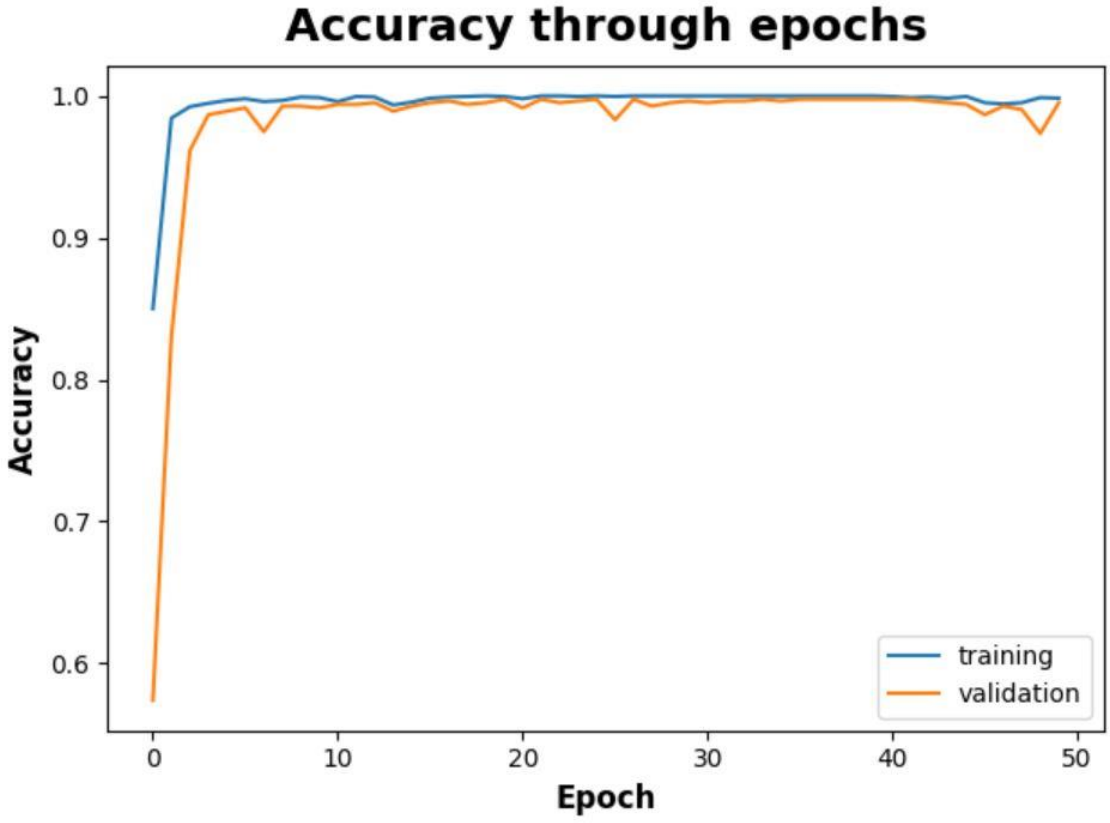

Figure 31 *– accuracy through epochs*

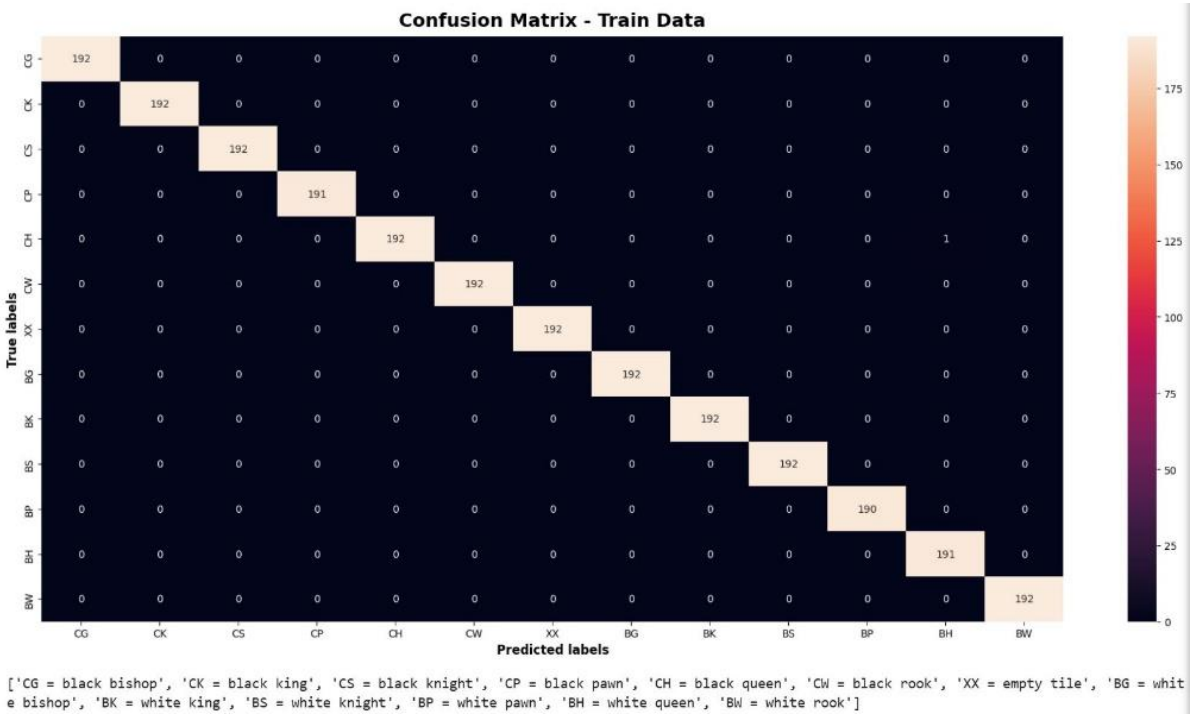

Figure 32 *– the confusion matrix – train data*

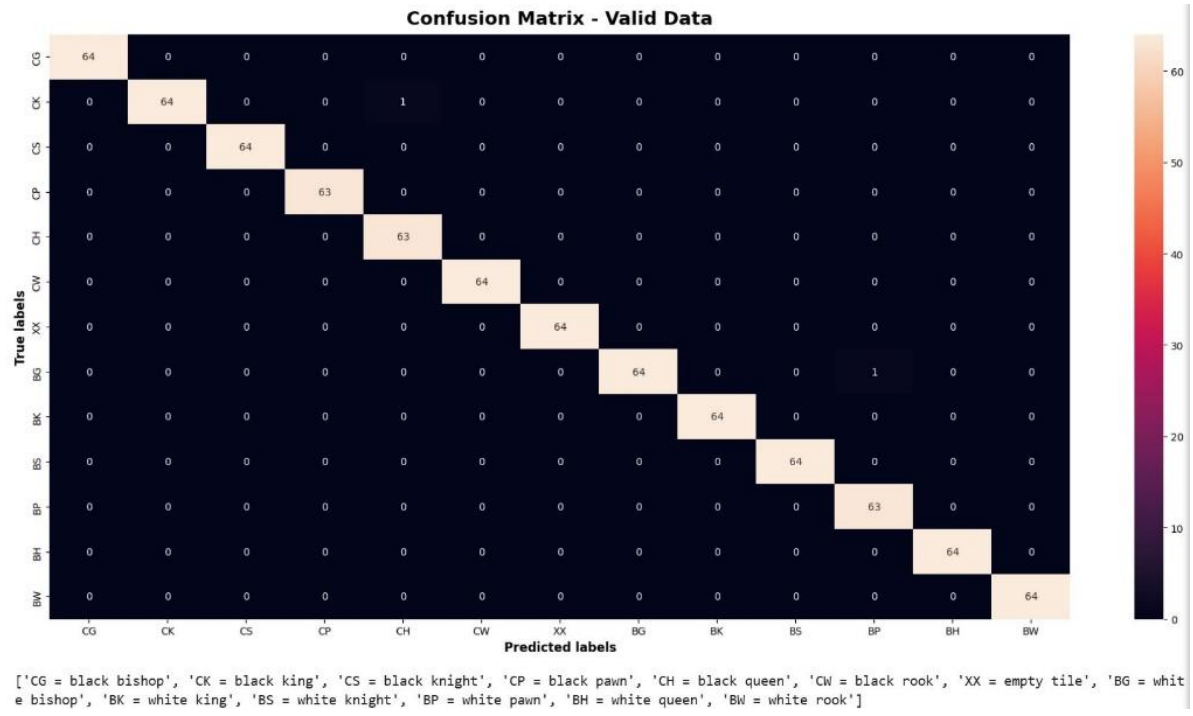

Figure 33 *– the confusion matrix – validate data*

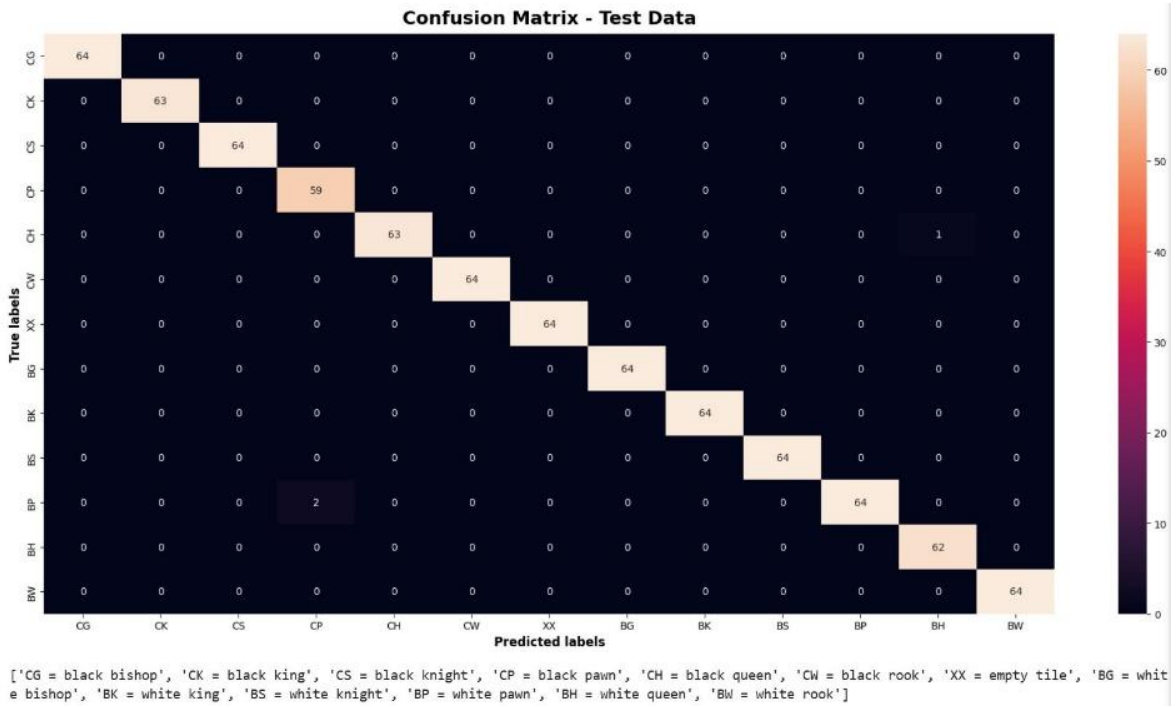

Figure 34 *– the confusion matrix – test data*

The total accuracy of the prepared CNN across all sets was above 0.966. The lowest result was obtained on the test set. These results clearly show that the neural network performs very well in recognizing chess pieces and can be reliably used in the project.

We also attach the notebook that we used for the CNN training to the report.

4. Conclusions

- The system successfully detects and classifies chess pieces using image processing and the trained CNN. It works in real-time and provides an accurate board representation.
- During the board recognition stage, all pieces must be removed from the chessboard. This is necessary for the Canny Edge Detection algorithm to correctly identify the grid lines of the board.
- At first, the dataset for training the neural network was prepared using photos of chess pieces printed in 3D. However, this approach encountered significant issues. Pieces such as bishops and pawns were difficult to distinguish in top-down images, and when the camera was positioned beside the board, taller pieces could obscure shorter ones. As a result, some pieces became invisible and could not be correctly classified by the network. For this reason, the decision was made to use 2D-printed chess pieces instead.
- The network was trained using a GPU, which significantly reduced computation time.
- Lighting conditions turned out to be a critical factor affecting recognition performance. For the network to function properly, the lighting had to remain consistent throughout the game. This requirement posed a major challenge when designing the testing setup.

References

- TheAILerner (https://theailearner.com/tag/cv2-getperspectivetransform/, https://theailearner.com/tag/cv2-warpperspective/) – perspective change – 2D homography (method and functions)
- the CLAHE algorithm: (https://www.youtube.com/watch?v=tn2kmbUVK50&t=204s&ab_channel=KevinWood%7CRobotics%26AI) – YouTube video
- Kaggle - "Detecting Chess Pieces with a CNN" - https://www.kaggle.com/code/thomassvisser/detecting-chess-pieces-with-a-cnn (Apache 2.0 open-source license)
- chess pieces ("Designed by brgfx / Freepik")
- ChatGPT was used to interpret new algorithms and to help building the project.
- OpenCV and TensorFlow documentation
- the lectures