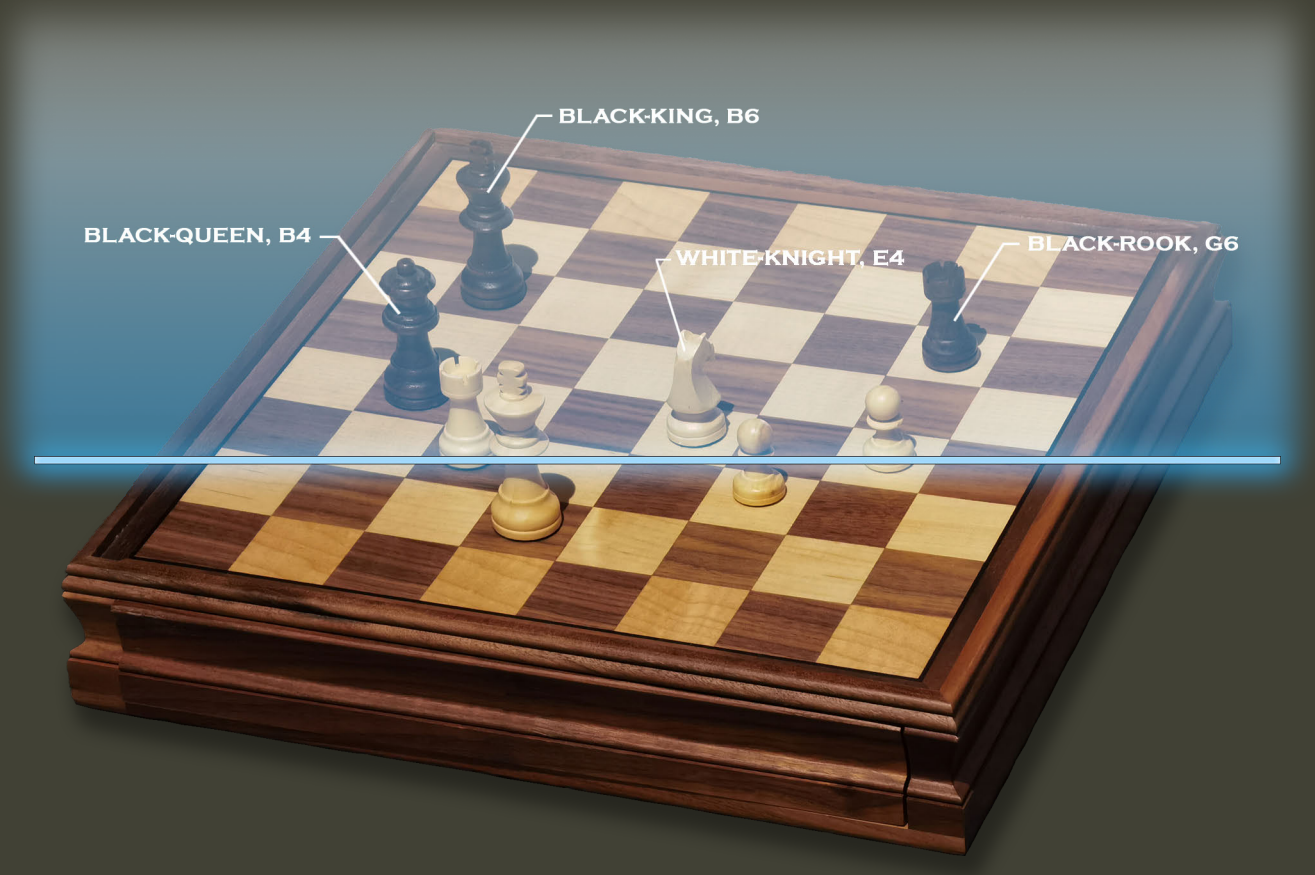
End-to-End Chess Recognition

Athanasios Masouris



End-to-End Chess Recognition

by

Athanasios Masouris

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Thursday August 31, 2023 at 10:00.

Student number:	5538009	
Project duration:	November 28, 2022 – August 31, 2023	
Thesis committee:	Dr. ir. J. van Gemert,	TU Delft, supervisor
	Dr. ir. S. Verwer,	TU Delft, external committee member

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This report documents the work of my thesis, titled “End-to-End Chess Recognition”, and it marks the completion of my master’s education in Computer Science at the Electrical Engineering, Mathematics & Computer Science (EEMCS) faculty of TU Delft. Thus, I would like to express my sincere gratitude to everyone who supported me throughout not only the preparation of this thesis but also my 2-year studies in the Netherlands.

The project was conducted within the Computer Vision Lab of TU Delft, under the supervision of Dr. Jan van Gemert. Therefore, I would like to convey my appreciation towards Jan, for the guidance and support he provided throughout this project. His feedback shaped the direction of my research. Furthermore, I would like to thank Dr. Sicco Verwer for honoring me with his presence on the thesis committee.

On a personal level, I am deeply grateful to my family, my father Panagiotis, my mother Katerina, and my brother Giannis, for their unwavering support and encouragement throughout the past two years. Finally, I would like to thank my friends for accompanying me during this journey, regardless of being afar, back in Athens, or with me here in Delft.

Athanasios Masouris
Delft, August 2023

Abstract

Chess recognition refers to the task of identifying the chess pieces configuration from a chessboard image. Contrary to the predominant approach that aims to solve this task through the pipeline of chessboard detection, square localization, and piece classification, we rely on the power of deep learning models and introduce two novel methodologies to circumvent this pipeline and directly predict the chessboard configuration from the entire image. In doing so, we avoid the inherent error accumulation of the sequential approaches and the need for intermediate annotations.

Furthermore, we introduce a new dataset, *Chess Recognition Dataset (ChessReD)*, specifically designed for chess recognition that consists of 10,800 images and their corresponding annotations. In contrast to existing synthetic datasets with limited angles, this dataset comprises a diverse collection of real images of chess formations captured from various angles using smartphone cameras; a sensor choice made to ensure real-world applicability. We use this dataset to both train our model and evaluate and compare its performance to that of the current state-of-the-art. Our approach in chess recognition on this new benchmark dataset outperforms related approaches, achieving a board recognition accuracy of 15.26% ($\approx$7x better than the current state-of-the-art).

Contents

1

Introduction

The game of chess has remained an object of fascination for researchers and enthusiasts alike throughout centuries. Due to its complexity, it requires a high degree of strategic thinking and problem-solving skills, testing the cognitive capabilities of not only the players but also the viewers. Professional chess players are renowned for their ability to anticipate moves in advance, and thus act on that information. Yet, reasoning the players' moves is a challenging task for the average viewer. While there are chess engines that leverage decision-making techniques and can explain the dynamics behind each move, they require the state of the game to be inputted. One option would be to manually enter the positions of the pieces on the engine's digital chessboard. A better alternative is *chess recognition*.

Chess recognition refers to the task of identifying the chess pieces configuration from a chessboard image, or alternatively *"given an image of a chessboard, output the type and the chessboard position of every piece in the image"*. The applications of this task span multiple domains, including not only the aforementioned use for automated game analysis, but also augmented reality gaming, educational tools (*e.g*. chess tutoring), and chess database cataloging, among others. Nevertheless, the success of the proposed methods depends on their ability to overcome several challenges related to chess recognition. The occlusions between pieces, the lighting conditions, the perspective variations, the reflections and the shadows, and the similarity between pieces represent some of the main concerns.

Another significant challenge for chess recognition is the lack of a comprehensive dataset that would facilitate research in the field. To address this issue, in this work, we introduce a new dataset specifically designed for chess recognition, named *Chess Recognition Dataset (ChessReD)*, that comprises a diverse collection of 10,800 images of chess formations. Leveraging this new dataset and the power of deep learning, we also introduce two novel methodologies for tackling this task. Chapter 2 presents these methodologies and the results of our approach in a scientific article.

The rest of the thesis is structured as follows. Chapter 3 provides insights into image processing techniques used by related works for chess recognition to exploit the geometric nature of the chessboard and identify its position in the image. Chapter 4 focuses on deep learning techniques and neural networks. Sections about convolutional neural networks (CNNs) and transformers are also included in this chapter to familiarize the reader with the ResNeXt (Section 4.2.5) and DETR (Section 4.3.5) model architectures used in the experiments of Chapter 2. Chapter 5 discusses in more detail the two datasets used in the aforementioned experiments, while Chapter 6 provides additional results and visualizations.

2

Scientific Article

End-to-End Chess Recognition

Athanasios Masouris
Delft University of Technology
Delft, The Netherlands
a.masouris@student.tudelft.nl

Jan van Gemert
Delft University of Technology
Delft, the Netherlands
j.c.vangemert@tudelft.nl

Abstract

Chess recognition refers to the task of identifying the chess pieces configuration from a chessboard image. Contrary to the predominant approach that aims to solve this task through the pipeline of chessboard detection, square localization, and piece classification, we rely on the power of deep learning models and introduce two novel methodologies to circumvent this pipeline and directly predict the chessboard configuration from the entire image. In doing so, we avoid the inherent error accumulation of the sequential approaches and the need for intermediate annotations.

Furthermore, we introduce a new dataset, Chess Recognition Dataset (ChessReD), specifically designed for chess recognition that consists of 10,800 images and their corresponding annotations. In contrast to existing synthetic datasets with limited angles, this dataset comprises a diverse collection of real images of chess formations captured from various angles using smartphone cameras; a sensor choice made to ensure real-world applicability. We use this dataset to both train our model and evaluate and compare its performance to that of the current state-of-the-art. Our approach in chess recognition on this new benchmark dataset outperforms related approaches, achieving a board recognition accuracy of 15.26% (≈7x better than the current state-of-the-art).

1. Introduction

Chess recognition from a single image poses a significant challenge in the field of computer vision. The task requires accurate identification of each chess piece's type and position on a chessboard configuration. The predominant approach [6, 15, 27, 29] is to divide the overall task into the tasks of chessboard detection, square localization, and piece classification within each square. However, this approach not only suffers from error accumulation throughout the intermediate steps but also often sets restrictions to the task, with regard to selected viewing angles [27]. In this paper, we propose a novel approach for chess recognition that circumvents this pipeline and directly predicts the positions of the pieces, with respect to the chessboard, from the entire image.

One key advantage of our approach is that it does not impose any restrictions on the input image, such as specific angles and orientations. Our model can handle images taken from various perspectives, enabling robust chess recognition. By leveraging a deep neural network, the model is able to extract and use relevant visual features to efficiently predict the pieces' type and positions. Additionally, our approach does not require any human input beyond the input image itself. Unlike traditional methods [8, 16, 21, 27] that rely on user input, such as manually selecting the corners of the chessboard or defining the player's perspective, our approach directly learns to recognize the chessboard and infer the piece positions strictly from the visual information in the input image.

Furthermore, a factor that has hindered research on this particular domain is the lack of a comprehensive real-world dataset. To this end, we introduce a new real-world dataset specifically designed for chess recognition, *Chess Recognition Dataset (ChessReD)*, consisting of $10,800$ images and their corresponding annotations, that would allow us to effectively train and evaluate our proposed approach. The dataset comprises a diverse collection of chessboard images, covering various viewing angles, lighting conditions, camera specifications, and piece configurations. By providing this dedicated dataset, we enable both further research in chess recognition algorithms and a common benchmark for researchers to evaluate their approaches.

The results indicate that our approach in chess recognition on this new benchmark dataset outperforms related approaches, achieving a board recognition accuracy of 15.26% (≈7x better than the current state-of-the-art). Thus, our contributions can be summarized as follows.

- We introduce the first dataset of real images for chess recognition, with increased variability in terms of viewing angles and chess formations compared to synthetic alternatives

- We demonstrate that chess recognition can be tackled using an end-to-end approach, with improved performance compared to sequential alternatives

2. Related Work

2.1. Early Approaches in Chess Recognition

Chess recognition has been a subject of research in the field of computer vision, with several approaches proposed to tackle the challenges associated mainly with detecting the chessboard, but also with recognizing the individual pieces on top of it. Early attempts in the field primarily focused on integrating the chess recognition task as a part of a chess-playing robotic system [1, 14, 21, 23]. These systems detected chess moves by comparing the previous frame, with a known game state, to the current frame. They relied on detecting the occupied squares of the chessboard along with the colors of the pieces occupying them. As such, these methods were only able to detect valid chess moves and failed to detect events when two pieces of the same color were swapped, either illegally, or by promoting pawns to another piece type. Additionally, they worked under the assumption that the previous inferred state is correct. Thus, in case of an erroneous move prediction, all of the subsequent game states would be incorrect. Despite the aforementioned issues, the same approach has been adopted by several studies [3, 4, 11, 12, 16, 24] since, with consecutive frames obtained also from a video stream [12, 21, 24]. In this study, contrary to these approaches, we aim to develop a robust method that does not rely on the correctness of the previous inferred state but rather performs chess recognition from a single input image.

2.2. Chessboard Detection

For cases when the previous state is unknown, chess recognition from a single image has also been the focus of studies. Same as with the previously mentioned approaches, the first step is to employ image processing techniques to detect the chessboard and the individual squares; a challenging task even on its own. While it can be simplified by explicitly asking the user to select the four corner points [8, 16, 21], modifying the chessboard [1, 7, 23] (*e.g*. using a reference frame around the chessboard), or setting constraints on the camera view angle (*e.g*. top-view) [1, 12, 15, 16, 21, 23, 24], Neufeld *et al*. [17] recognized that these approaches do not represent a general solution, where the chessboard could be in arbitrary locations or the image taken from various camera angles. They proposed a line-based detection method which they combined with probabilistic reasoning. However, they also restricted the setting by expecting the camera angle to be in the range of a human player's perspective. Other studies have also exploited specific viewing angles, such as the players' perspectives [3, 4, 8, 27] or side views [7, 18]. While chessboard detection utilizing the Harris corner detection algorithm [1, 11, 12], template matching [14, 23], or flood fill [24] have been explored, in accordance with [17], line-based chessboard detection methods have received significant research attention [3, 4, 7, 15, 22, 29, 30]. Czyzewski *et al*. [6] introduced an approach based on iterative heat map generation which visualizes the probability of a chessboard being located in a sub-region of the image. After each iteration, the four-sided area of the image containing the highest probability values is cropped and the process is repeated until convergence. While this method involves a great computational overhead, it is able to detect chessboards from images taken from varied angles, with poor quality, and regardless of the state of the actual chessboard (*e.g*. damaged chessboard with deformed edges), with a 99.6% detection accuracy. Wölflein and Arandjelović [27] proposed a chessboard detection method that leveraging the geometric nature of the chessboard, utilizes a RANSAC-based algorithm to iteratively refine the homography matrix and include all the computed intersection points. Their method demonstrated impressive results, since it successfully detected all of the chessboards in their validation dataset. However, it's worth noting that the dataset only included images with viewing angles within the range of a player's perspective. In our paper, we bypass the chessboard detection task, allowing the deep learning models to internally infer its position, and thus we do not rely on user input, or specific viewing angles.

2.3. Piece Classification

Upon detection of the chessboard, the next step the aforementioned approaches employ is piece classification. A number of techniques have been developed to address this task, either in a 2-way approach (*i.e*. color and type), or 1-way by treating each combination of piece color and type as a separate class (*e.g*. "white-rook"). In Matuszek *et al*. [14], the authors utilized one classifier to determine the piece color, and then for each color they trained a type classifier using concatenated scale-invariant feature transform (SIFT) and kernel descriptors for features. A similar approach was used in Ding [8], where the author employed SIFT and histogram of oriented gradients (HOG) as feature descriptors for piece type classification with support vector machine (SVM) classifiers. The color was subsequently detected by comparing the binarized image of the square with that of an empty one. Danner and Kafafy [7] and Xie *et al*. [29] argued that the lack of distinguishable texture in small objects, such as chess pieces, leads to insufficient features obtained using SIFT descriptors. Both studies suggested a template matching approach for piece classification, comparing the contours of the detected pieces with reference templates obtained from various angles. Wei *et al*. [25] proposed an approach to recognize pieces using a depth camera and a vol-

umetric convolutional neural network (CNN). More recent studies [6, 15, 18, 27] follow the 1-way approach for piece classification. They train CNNs to distinguish between 12 or 13 classes of objects (*i.e*. six piece types in both colors and one for empty squares), obtaining impressive results. In Czyzewski *et al*. [6], they also leverage domain knowledge, to improve piece classification, by utilizing a chess engine to calculate the most probable piece configurations and clustering similar figures into groups to deduce formations based on cardinalities. Additionally, given the variation in appearance between chess sets, Wölflein and Arandjelović [27] proposed a novel fine-tuning process for their piece classifier to unseen chess sets. In our paper, same as with the chessboard detection task, the classification of the pieces is performed by the deep learning model, without the need to train a separate piece classifier.

2.4. Chess Datasets

A common problem frequently mentioned in literature [6, 8, 15, 27] is the lack of a comprehensive chess dataset. This issue hinders not only the ability to fairly evaluate the proposed methods in a common setting but also impedes the deployment of deep learning end-to-end approaches that require a vast amount of data. One proposed solution to this problem is the use of synthetic generated data. In [25], the authors produce point cloud data using a 3D computer-aided design (CAD) model, while Blender [5] was used to produce synthetic image datasets from a top view camera angle [16], or the player's perspective [27]. In our paper, we introduce the first chess recognition dataset of real images, without setting any of the aforementioned restrictions regarding the viewing angles.

3. Chess Recognition Dataset (ChessReD)

The availability of large-scale annotated datasets is critical to the advancement of computer vision research. In this section, we tackle a main issue in the field of chess recognition (*i.e*. the lack of a comprehensive dataset) by presenting a novel dataset specifically designed for this task. The dataset comprises a diverse collection of images of chess formations captured using smartphone cameras; a sensor choice made to ensure real-world applicability.

3.1. Data Collection and Annotation

The dataset was collected by capturing images of chessboards with various chess piece configurations. To guarantee the variability of those configurations, we relied upon the chess opening theory. The Encyclopedia of Chess Openings (ECO) classifies opening sequences into five volumes with 100 subcategories each that are uniquely identified by an ECO code. We randomly selected 20 ECO codes from each volume. Subsequently, each code of this set was randomly matched to an already played chess game that followed the particular opening sequence denoted by the ECO code; thus creating a set of 100 chess games. Finally, using the move-by-move information provided by Portable Game Notations (PGNs) that are used to record chess games, the selected games were played out on a physical chessboard with images being captured after each move.

Three distinct smartphone models were used to capture the images. Each model has different camera specifications, such as resolution and sensor type, that introduce further variability in the dataset. The images were also taken from diverse angles, ranging from top-view to oblique angles, and from different perspectives (*e.g*. white player perspective, side view, etc.). These conditions simulate real-world scenarios where chessboards can be captured from a bystander's arbitrary point of view. Additionally, the dataset includes images captured under different lighting conditions, with both natural and artificial light sources introducing these variations. Most of these variations are illustrated in the four image samples of Figure 1. Each of those samples highlights a different challenge in chess recognition. Occlusions between pieces occur more often in images captured from a low angle (Fig. 1c) or a player's perspective (Fig. 1b), while pieces are rarely occluded in top-view images (Fig. 1d). However, distinct characteristics of pieces (*e.g*. the queen's crown) that could aid the chess recognition task are less distinguishable in a top-view.

The dataset is accompanied by detailed annotations providing information about the chess pieces formation in the images. Therefore, the number of annotations for each image depends on the number of chess pieces depicted in it. There are 12 category ids in total (*i.e*. 6 piece types per color) and the chessboard coordinates are in the form of algebraic notation strings (*e.g*. "a8"). These annotations were automatically extracted from Forsyth-Edwards Notations (FENs) that were available to us by the games' PGNs. Each FEN string describes the state of the chessboard after each move using algebraic notation for the piece types (*e.g*. "N" is knight) , capitalization for the piece colors (*i.e*. white pieces are denoted with uppercase letters, while black pieces with lowercase letters), and digits to denote the number of empty squares. Thus, by matching the captured images to the corresponding FENs, the state of the chessboard in each image was already known and annotations could be extracted. To further facilitate research in the chess recognition domain, we also provide bounding-box and chessboard corner annotations for a subset of 20 chess games. An annotated sample is presented in Figure 2. The different colors for the corner points represent the four distinct corner annotations (*i.e*. bottom-left, bottom-right, top-left, and top-right) that are relative to the white player's perspective. For instance, the corner annotated with the red color in Figure 2 is a *bottom-left* corner. The discrimination between these different types of corners provides information about

the orientation of the chessboard that can be leveraged to determine the image's perspective and viewing angle.

(a) Corner view (b) Player view

(c) Low angle (d) Top view

Figure 1. Image samples from ChessReD

3.2. Data Statistics

In this section, we present an overview of the ChessReD statistics. As mentioned in Section 3.1, the dataset consists of 100 chess games, each with an arbitrary number of moves and therefore images, amounting to a total of 10,800 images being collected. The dataset was split into training, validation, and test sets following an 60/20/20 split. Since two consecutive images of a chess game differ only by one move, the split was performed on game-level to ensure that quite similar images would not end up in different sets. The split was also stratified over the three distinct smartphone cameras (Apple iPhone 12, Huawei P40 pro, Samsung Galaxy S8) that were used to capture the images. Table 1 presents an overview of the image statistics per smartphone. The three smartphone cameras introduced variations to the dataset based on the distinct characteristics of their sensors. For instance, while the image resolution for the Huawei phone was 3072x3072, the resolution for the remaining two models was 3024x3024.

Table 2 presents an overview of the annotations in ChessReD. In Table 2a we can see that there is a significant imbalance between provided annotations for the piece type "Pawn" and the rest of the types in the dataset. This was to be expected since every chess game starts with 8 pawns in each side and only one or two of the remaining piece types.

Smartphone	**Number of images**		
	Train	**Val**	**Test**
Apple iPhone 12	2,146	851	638
Huawei P40 pro	2,102	638	871
Samsung Galaxy S8	2,231	703	620
Total	**6,479**	**2,192**	**2,129**

Table 1. Overview of the image statistics

Figure 2. Bounding box and corner point annotations in ChessReD2K

Regarding the colors of the pieces, no imbalance is detected in the dataset. Additionally, while annotations about the position of the pieces in algebraic notation are available for every image in the dataset, we provide bounding box and chessboard corner annotations only for a subset of 20 randomly selected games from the train, validation, and test sets. For this subset we followed a 70/15/15 split stratified over the smartphone cameras, which led to a total of 14 training games (1442 images), 3 validation games (330 images), and 3 test games (306 images) being annotated. In Table 2b we can see an overview of the annotation statistics for this subset, named ChessReD2K.

4. End-to-End Chess Recognition

Unlike the conventional pipeline in chess recognition that involves chessboard detection, square localization, and chess piece classification, the focus of this study was to develop an end-to-end approach that tackles the recognition task utilizing only a single image as input. Thus, the de-

Piece type	Number of instances					
	Train		Val		Test	
	Black	White	Black	White	Black	White
Pawn	35,888	35,021	11,410	11,042	11,616	11,472
Rook	9,317	9,260	2,605	2,876	2,992	3,077
Knight	6,158	6,471	2,222	2,206	2,032	2,202
Bishop	6,681	6,768	2,167	2,003	2,301	2,067
Queen	4,076	3,996	1,011	1,013	1,145	1,109
King	6,479	6,479	2,192	2,192	2,129	2,129

(a) Piece positions on the chessboard in ChessReD

Piece type	Number of instances					
	Train		Val		Test	
	Black	White	Black	White	Black	White
Pawn	8,059	7,653	1,511	1,625	1,719	1,624
Rook	2,293	2,250	471	447	433	433
Knight	1,276	1,423	178	274	278	278
Bishop	1,578	1,607	380	335	296	304
Queen	862	838	125	126	157	160
King	1,442	1,442	330	330	306	306

(b) Bounding boxes in ChessReD2K

Table 2. Overview of the annotation statistics

veloped method should take as input an image of a chessboard and output the type and the positions of the pieces relative to the board. To this end, we experimented with two different solutions by treating the problem either as a multi-class multi-label classification or as a *relative object detection* task.

4.1. Classification Approach

In the classification approach, we treat each chessboard square as a distinct label. Since there are 64 squares in each image, and thus 64 labels, this problem is treated as a multi-label classification task. Additionally, each square in the chessboard is either unoccupied or occupied by one of the 12 different types of pieces (*i.e*. 6 per color) in chess. Therefore, to each label we assign one of 13 classes (*i.e*. 12 piece types and empty), constituting this approach multi-label multi-class classification. By formulating this approach as such, the goal is for the model to learn the intricate relationships and visual patterns associated with the individual squares.

4.2. Relative Object Detection Approach

In addition to the multi-class multi-label classification approach, we explore a novel technique for chess recognition which we call *relative object detection*. Contrary to conventional object detection methods that predict bounding box coordinates in terms of absolute position in the image frame, our modified method aims to predict the x and y coordinates of the objects relative to the chessboard grid in the image. In this manner, discrete coordinates that align with the chessboard positions are used to provide spatial information of its layout. For instance, the relative position (0,0) corresponds to the chessboard square denoted by "a8" in chess algebraic notation. Furthermore, since we only need to predict the relative coordinates, we can omit the height and width estimation, effectively bypassing the complexities of the bounding box size estimation.

5. Implementation Details

5.1. Classification

For the classification approach, we employed a ResNeXt [28] model. Based on the ResNet [9] architecture, ResNeXt models constitute a powerful variant that can learn complex representations of images. Their introduced concept of "cardinality" (*i.e*. the number of parallel branches used in each residual block) both enables deeper architectures with reduced computation complexity and allows complex representations to be learned by aggregating the information of the multiple branches. Because of these modifications, this family of models can achieve impressive results in image classification.

For our experiments, we trained the *resnext101_32d* variant, which uses a cardinality of 32 and a width (*i.e*. number of filters) of 8. This means that each residual block in the network contains 32 parallel convolutional layers, and each of them has a width of 8. Additionally, there are 101 layers in the network, which amount to a total of 88.8M parameters. We trained this model from scratch for 200 epochs, with early stopping enabled and a batch size of 8 samples, using a cross-entropy loss function. We used an Adam [10] optimizer with a learning rate of 0.001, which was reduced to 0.0001 after the 100th epoch. The same training recipe was used to train ResNext on all of the datasets in our experiments (Sec. 6).

5.2. Relative Object Detection

As mentioned in Section 4.2, the goal is to predict a set of coordinates x and y for the chess pieces relative to the chessboard grid in the image. Thus, traditional object detection models that use Region Proposal Networks (RPNs) [20] or anchor boxes [19] are not suitable for this task, since they output absolute image coordinates. However, a single end-to-end object detection model, like Detection Transformer (DETR) [2], that directly predicts bounding boxes and class labels for objects in an image could be employed. DETR uses a transformer encoder-decoder architecture, with the encoder taking as input a feature map produced by a convo-

lutional backbone network and the decoder generating the final predictions using self-attention mechanisms to attend to different parts of the feature map.

For our experiments, we attempted to train a modified version of DETR that predicts relative object coordinates and omits the height and width dimensions for the bounding boxes of the traditional object detection task. ResNext101_32d was used as a backbone network for feature extraction. We set the number of queries (*i.e*. the maximum number of objects that DETR can detect in an image) to 32, since each chessboard can have at most 32 chess pieces on top of it. DETR also requires a separate class for "background", which in our case corresponds to "empty" squares. Thus, the number of classes that the model is trained to predict is 13 (*i.e*. 12 piece types and background). We trained the model from scratch for a total of 800 epochs, with early stopping enabled and a batch size of 8 samples, using DETR's default bipartite matching loss for set predictions, which takes into account both the class prediction and the similarity of the predicted and ground truth coordinates. We used an AdamW [13] optimizer with separate learning rates for the backbone network and the encoder-decoder architecture. In particular, the initial learning rates were set to 10^{-5} and 10^{-6} for the encoder-decoder and backbone, respectively, and a scheduler was used to reduce both by a factor of 10 every 300 epochs. Furthermore, gradient clipping was used with a threshold of 0.1.

However, the training of this modified DETR variant for chess recognition did not yield optimal results, with the model being unable to successfully detect chess pieces in the images of ChessReD. This issue could potentially be linked to DETR's inherent limitation in detecting small objects [2, 31], especially when considering the intricacies of the dataset (*e.g*. occlusions) and the relatively small sizes of individual pieces. Due to the unsuccessful convergence of the DETR variant, it will not be used in the experiments of Section 6. Nevertheless, end-to-end relative object detection with transformers is a promising area that should be further investigated, with the focus being on refining the model architecture ([31]) or the training objective.

6. Experiments

6.1. Exp1: Comparison with the state-of-the-art

To the best of our knowledge, the current state-of-the-art approach in chess recognition, namely *Chesscog*, was introduced in Wölflein and Arandjelović [27]. In their experiments, Chesscog achieved a 93.86% accuracy in chess recognition on a synthetic dataset [26] rendered in Blender [5], with a 0.23% per-square error rate. Additionally, the authors introduced a few-shot transfer learning approach to unseen chess sets and the system demonstrated a 88.89% accuracy and 0.17% per-square error rate, when tested on a set of previously unseen images of chessboards. In this section, we will compare the performance of our approach with that of Chesscog's, both on their Blender dataset and on our newly introduced ChessReD.

6.1.1 Current SOTA: Chesscog

Chesscog attempts to solve the chess recognition task using a pipeline that involves chessboard detection, square localization, occupancy classification, and piece classification. It leverages the geometric nature of the chessboard to detect lines and employs a RANSAC-based algorithm to compute a projective transformation of the board onto a regular grid. Subsequently, individual squares are localized based on the intersection points and an occupancy classifier is used on each individual square. Finally, the pieces on the occupied squares are classified into one of 12 classes, using a pre-trained piece classifier. The piece classifier is used on image patches of the squares that are cropped based on a heuristic approach that extends the bounding boxes based on the square's location on the chessboard. It is also important to mention that during inference the user must manually input the specific player's perspective (*i.e*. "white" or "black") to determine the orientation of the board.

6.1.2 How does the classification approach compare to Chesscog on the synthetic Blender dataset?

First, we evaluate the performance of our classification approach on the Blender dataset and compare it with that of Chesscog. The Blender dataset comprises a set of 4,888 synthetic chessboard images with distinct piece configurations, multiple lighting conditions, a limited range of viewing angles (between 45° and 60° to the board), and images taken only from the players' perspectives. We trained our ResNeXt model following the recipe described in Section 5.1 on the dataset's training samples. Subsequently, we evaluated our trained model's performance on the test set. The first two columns of Table 3 demonstrate the evaluation results for both approaches on the Blender dataset. We use the same evaluation metrics as in Wölflein and Arandjelović [27].

Chesscog outperforms our classification approach across all metrics. For the percentage of boards with no mistakes, which reveals a model's ability to achieve perfect board recognition, Chesscog demonstrates a significant advantage with 93.86% of boards correctly predicted, while ResNeXt achieves this only in 39.76% of the boards. When one mistake is allowed per board prediction, Chesscog can successfully recognize almost all of the boards, with ResNeXt's performance improving significantly and reaching 65.2%. Chesscog's superiority is also corroborated by the substantially lower mean number of incorrect squares

per board (0.15 vs. 1.19 for ResNeXt) and per-square error rate (0.23% vs. 1.86% for ResNeXt).

6.1.3 How does the classification approach compare to Chesscog on the real ChessReD dataset?

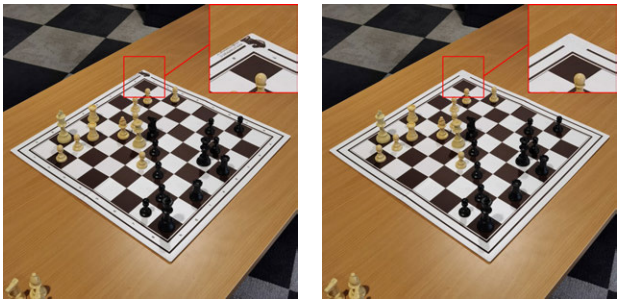
In this section, we compare the performance of our approach with that of Chesscog's on our ChessReD. We trained our ResNeXt model, using again the recipe of Section 5.1, and finetuned the Chesscog classifiers as mentioned in [27], using two images of the starting position from both players' perspectives. Furthermore, for a fair comparison we needed to take into account that Chesscog cannot infer the orientation of the chessboard and requires for it to be manually inputted. Since this information is not available in our dataset, we address it by generating all possible orientations for the detected chessboards during evaluation.

Both approaches faced increased challenges when tested on ChessReD, resulting in a performance drop across all metrics, as seen in Table 3. While our ResNeXt model can still demonstrate competitive results, recognizing successfully 15.26% of boards with no mistakes and 25.92% of boards with less than one mistake, Chesscog's accuracy decreases significantly, achieving only 2.3% and 7.79% in these metrics, respectively. Chesscog's performance deterioration is also evident by its 42.87 incorrect squares per board on average and the 73.65% per-square error rate. ResNeXt's performance for these metrics was 3.40 and 5.31%, respectively.

Upon further investigation, one important factor that led to Chesscog's performance degradation was the inaccurate results of the chessboard detection process and the accumulation of the error throughout the pipeline. While the limited range of angles present in the Blender dataset of the previous section enabled Chesscog to achieve 100% accuracy in chessboard detection, the corresponding accuracy in our dataset is 34.38%. This issue highlights the sensitivity of the image processing algorithms employed for chessboard detection to their hyperparameters and the necessity to fine-tune them across different datasets.

To further compare the performance of both approaches, we conducted the same evaluation without taking into account the failed chessboard detections by Chesscog. In the last two columns of Table 3, we evaluate the performance of both approaches on the subset of the ChessReD's test set (denoted as ChessReD*) consisting of the 34.38% (732) of the images in which Chesscog was able to detect the chessboard. While Chesscog's performance shows significant improvement when we don't consider those erroneous chessboard detections, it remains inferior in comparison to the results achieved by our classification approach across all metrics.

6.2. Exp2: Does the classification approach rely on chessboard marks for recognition?



(a) With marks (b) Without marks

Figure 3. Sample pair of images for the ablation study

One significant advantage of our approaches is that they do not require any further input to determine the orientation of the chessboard in an image. Yet, the visual cues used by the models to deduce the chessboard's orientation remain unclear. To this end, we conducted an ablation study to investigate whether the ResNeXt model relies on specific marks of the chessboard (*e.g.* bishop logo in Fig. 3a) to determine the board's orientation and subsequently recognize the chess configuration.

We hypothesize that the necessity of those marks for successful chess recognition increases with the number of moves that have been made prior to capturing the image. The intuition behind this hypothesis is that in the early game of chess, the majority of the pieces remain in their starting position, so determining the boards orientation poses less of a challenge, while in the end game, only a few pieces remain on the board and they are usually far from their starting position. To validate this hypothesis, we created a dataset consisting of 30 test images that were randomly selected from the subset of images that the model was able to successfully recognize in the evaluation of Section 6.1.3. The test images were evenly distributed across three categories: early-game, mid-game, and end-game. These categories correspond to images that were taken when less than 30, more than 30 but less than 75, or more than 75 moves had been made prior to capturing the images, respectively. Subsequently, we manually removed the marks, such as the bishop logo or the algebraic chess notation on the sides, and evaluated again the performance of ResNeXt on this subset of 30 modified images. A sample pair of images is illustrated in Figure 3. The model achieved an overall accuracy (*i.e.* boards with no mistakes) of 66.6% on this subset, with a perfect recognition in the early-game images, 60% accuracy in mid-game images, and 40% accuracy in end-game images.

Metric	Blender Dataset		ChessReD		ChessReD*	
	Chesscog	ResNeXt	Chesscog	ResNeXt	Chesscog	ResNeXt
Mean incorrect squares per board	**0.15**	1.19	42.87	**3.40**	12.96	**3.35**
Boards with no mistakes (%)	**93.86%**	39.76%	2.30%	**15.26%**	6.69%	**15.30%**
Boards with < 1 mistake (%)	**99.71%**	65.20%	7.79%	**25.92%**	22.67%	**27.04%**
Per-square error rate (%)	**0.23%**	1.86%	73.64%	**5.31%**	39.57%	**5.24%**

Table 3. Performance evaluation for Chesscog's and our classification approach's (ResNeXt) predictions on the corresponding test sets. *ChessReD** represents the subset of the test images in which Chesscog could detect the chessboard.

7. Discussion

The evaluation on the Blender dataset revealed that Chesscog outperforms our classification approach. However, further experimentation on our newly introduced ChessReD showcased a shift in both methods' performances, with ResNeXt surpassing Chesscog across all metrics. It is evident that Chesscog's low chessboard detection rate (34.38%), which is attributed to the diverse angles and occlusions introduced by our dataset, significantly contributed to that shift, while the specific range of angles used in the Blender dataset, enabled Chesscog to successfully detect the chessboard in all cases and achieve a remarkable end-to-end performance.

The ablation study provided significant insights into our ResNeXt model's reliance on specific marks for determining chessboard orientation, and therefore chess recognition. The study confirmed our hypothesis that the necessity of those marks increases with the number of moves made prior to image capture. The model achieved higher accuracy in early-game images, where most of the pieces remained in their starting positions, and lower accuracy in end-game images, where only a few pieces were still on the board and farther from their starting positions. While depending on such marks could be challenging in cases where they are absent or obscured, it could prove to be an advantage in end-game states in which even human annotators can have trouble determining the board's orientation without them.

8. Limitations

While our study sheds light on the importance of end-to-end deep learning approaches for chess recognition, the limitations of these solutions should also be considered. An inherent weakness of the classification approach is its inability to recognize labels that are absent from the dataset that it was trained on. For instance, if a specific piece/square combination was first seen at inference time, the model would be unable to assign the corresponding label. On the other hand, the relative object detection approach would not encounter this issue, but as a transformer-based solution it's difficult to converge when trained on a small dataset. Finally, finetuning these models on previously unseen data would require considerably more resources and compute time compared to finetuning a simple CNN piece classifier in the sequential approaches.

Regarding ChessReD, although including a single chess set in the images was a design choice, this lack of diversity impedes the development of solutions with broader applicability. Yet, it is feasible to enhance the dataset by collecting varied data with relative positional annotations (*i.e.* FEN strings instead of bounding boxes) from chess tournaments recordings, where the players are obliged to annotate their every move.

9. Conclusion

Our experiments demonstrate the effectiveness of our classification approach in chess recognition tasks, while also revealing Chesscog's advantages on certain datasets. However, with the focus being on real-world applicability, the ChessReD dataset, consisting of real images with varied angles and perspectives, poses a more challenging benchmark for chess recognition, and thus the experimental results establish our approach as the state-of-the-art method for this task. Moving forward, improving the model's ability to generalize by either enhancing the dataset, or incorporating domain adaptation techniques, should be explored. Additionally, the relative object detection approach, if converged, may constitute a more robust solution for chess recognition, and thus requires further studying.

References

[1] Nandan Banerjee, Debal Saha, Atikant Singh, and Gautam Sanyal. A simple autonomous robotic manipulator for playing chess against any opponent in real time. In *Proceedings of the International Conference on Computational Vision and Robotics*, 2011. 2

[2] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020. 5, 6

[3] Andrew Tzer-Yeu Chen, I Kevin, and Kai Wang. Computer vision based chess playing capabilities for the baxter humanoid robot. In *2016 2nd International Conference on*

Control, Automation and Robotics (ICCAR), pages 11–14. IEEE, 2016. 2

[4] Andrew Tzer-Yeu Chen and Kevin I-Kai Wang. Robust computer vision chess analysis and interaction with a humanoid robot. *Computers*, 8(1):14, 2019. 2

[5] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018. 3, 6

[6] Maciej A Czyzewski, Artur Laskowski, and Szymon Wasik. Chessboard and chess piece recognition with the support of neural networks. *Foundations of Computing and Decision Sciences*, 45(4):257–280, 2020. 1, 2, 3

[7] Cheryl Danner and Mai Kafafy. Visual chess recognition. *URL: http://web. stanford. edu/class/ee368/Project_Spring_1415/Reports/Danner_Kafafy. pdf*, 2015. 2

[8] Jialin Ding. Chessvision: Chess board and piece recognition. In *Tech. rep*. Stanford University, 2016. 1, 2, 3

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 5

[10] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 5

[11] Paweł Kołosowski, Adam Wolniakowski, and Kanstantsin Miatliuk. Collaborative robot system for playing chess. In *2020 International Conference Mechatronic Systems and Materials (MSM)*, pages 1–6. IEEE, 2020. 2

[12] Can Koray, Emre Sumer, V Struc, et al. A computer vision system for chess game tracking. In *21st Computer Vision Winter Workshop, Rimske Toplice, Slovenia*, 2016. 2

[13] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 6

[14] Cynthia Matuszek, Brian Mayton, Roberto Aimi, Marc Peter Deisenroth, Liefeng Bo, Robert Chu, Mike Kung, Louis LeGrand, Joshua R Smith, and Dieter Fox. Gambit: An autonomous chess-playing robotic system. In *2011 IEEE International Conference on Robotics and Automation*, pages 4291–4297. IEEE, 2011. 2

[15] Anav Mehta. Augmented reality chess analyzer (archessanalyzer): In-device inference of physical chess game positions through board segmentation and piece recognition using convolutional neural network. *arXiv preprint arXiv:2009.01649*, 2020. 1, 2, 3

[16] Afonso de Sá Delgado Neto and Rafael Mendes Campello. Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 152–160. IEEE, 2019. 1, 2, 3

[17] Jason E Neufeld and Tyson S Hall. Probabilistic location of a populated chessboard using computer vision. In *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, pages 616–619. IEEE, 2010. 2

[18] David Mallasén Quintana, Alberto Antonio del Barrio García, and Manuel Prieto Matías. Livechess2fen: A framework for classifying chess pieces based on cnns. *arXiv preprint arXiv:2012.06858*, 2020. 2, 3

[19] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 5

[20] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015. 5

[21] Emir Sokic and Melita Ahic-Djokic. Simple computer vision system for chess playing robot manipulator as a project-based learning example. In *2008 IEEE International Symposium on Signal Processing and Information Technology*, pages 75–79. IEEE, 2008. 1, 2

[22] K.Y. Tam, J.A. Lay, and D. Levy. Automatic grid segmentation of populated chessboard taken at a lower angle view. In *2008 Digital Image Computing: Techniques and Applications*, pages 294–299, 2008. 2

[23] David Urting and Yolande Berbers. Marineblue: A low-cost chess robot. In *Robotics and Applications*, pages 76–81, 2003. 2

[24] Victor Wang and Richard Green. Chess move tracking using overhead rgb webcam. In *2013 28th International Conference on Image and Vision Computing New Zealand (IVCNZ 2013)*, pages 299–304. IEEE, 2013. 2

[25] Yu-An Wei, Tzu-Wei Huang, Hwann-Tzong Chen, and JenChi Liu. Chess recognition from a single depth image. In *2017 IEEE International Conference on Multimedia and Expo (ICME)*, pages 931–936. IEEE, 2017. 2, 3

[26] G Wölflein and O Arandjelovic. Dataset of rendered chess game state images. *Open Science Framework*, 2021. 6

[27] Georg Wölflein and Ognjen Arandjelović. Determining chess game state from an image. *Journal of Imaging*, 7(6):94, 2021. 1, 2, 3, 6, 7

[28] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017. 5

[29] Youye Xie, Gongguo Tang, and William Hoff. Chess piece recognition using oriented chamfer matching with a comparison to cnn. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 2001–2009. IEEE, 2018. 1, 2

[30] Youye Xie, Gongguo Tang, and William Hoff. Geometry-based populated chessboard recognition. In *Tenth International Conference on Machine Vision (ICMV 2017)*, volume 10696, pages 9–13. SPIE, 2018. 2

[31] Xizhou Zhu, Weijie Su, Lewei Lu, Bin Li, Xiaogang Wang, and Jifeng Dai. Deformable detr: Deformable transformers for end-to-end object detection. *arXiv preprint arXiv:2010.04159*, 2020. 6

3

Image Processing

Image processing is a fundamental field in computer vision that includes a wide range of techniques and tools for manipulating digital images. From simple image enhancement (*e.g.* deblurring) to complex object detection algorithms, image processing methods can be applied to a variety of computer vision tasks and successfully exploit meaningful visual information from images. In this chapter, we will delve into various image processing techniques commonly employed in chess recognition, including edge detection, line detection, line clustering, and homography matrices for projective transformations.

3.1. Edge detection

Edge detection is an important technique in image processing, as it serves as a foundation for higher-level methods. Qualitatively, edges occur at the boundaries of regions of the image with severe changes in color, intensity, or texture. These boundaries often correspond to object edges, constituting edge detection as a significant step in several computer vision tasks, such as object recognition.

Given the conditions under which edges occur, a reasonable approach for detecting them is to identify points, or pixels, in an image where there are rapid variations in intensity. These points indicate borders between regions of interest, which are in turn indicative of object boundaries. Exploiting this observation, various edge detection methods have been proposed over the years, with Sobel-Feldman operator (Sobel, Feldman, et al. 1968), Prewitt operator (Prewitt et al. 1970), and Canny edge detector (Canny 1986) being the most commonly used ones.

3.1.1. Sobel-Feldman operator

While being introduced in 1968, the Sobel-Feldman operator remains a fundamental tool for detecting edges. More commonly used on grayscale images, it is a simple but yet effective edge detection operator. It is based on the convolution of an image with two 3x3 masks, one for the horizontal and one for the vertical direction. The masks are designed to calculate the magnitude and the direction of the gradient at each pixel in the image. The horizontal K_x and vertical K_y kernels of the masks are presented in Equation (3.1).

$$K_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \quad K_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.1}$$

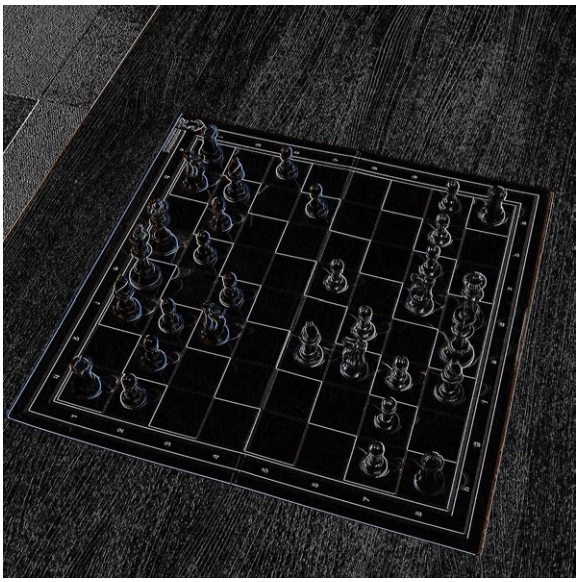
The gradient components in each direction, G_x and G_y, can be computed by convolving the image with the corresponding kernels K_x and K_y, respectively. Subsequently, the absolute gradient magnitude G at the pixel (x, y) is computed as the Euclidean norm of those gradients:

$$G(x, y) = \sqrt{{G_x}^2 + {G_y}^2}$$

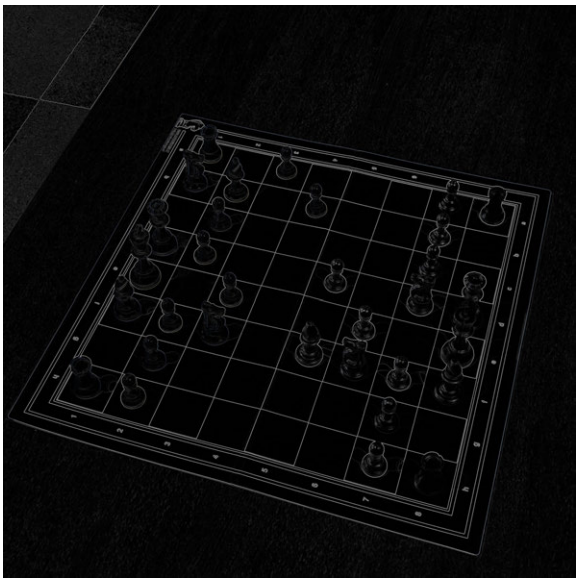
Figure 3.1 illustrates the results of applying the Sobel-Feldman operator on an image sample of the ChessReD.

(a) Original image

(b) Horizontal filter



(c) Vertical filter



(d) Magnitude

Figure 3.1: Results of applying the Sobel-Feldman filter on an image sample of ChessReD

3.1.2. Prewitt operator

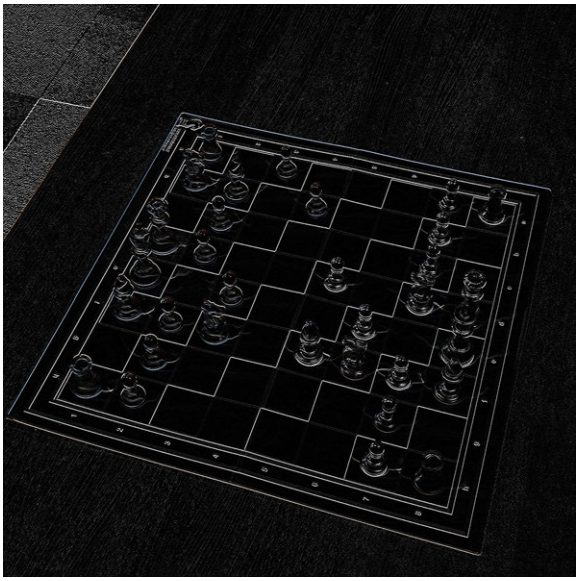
The Prewitt operator is quite similar to the Sobel-Feldman operator, with the only difference being the variation in the weights of the utilized horizontal and vertical masks. The Prewitt kernels are presented in Equation (3.2).

$$K_x = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix}, \quad K_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} \tag{3.2}$$

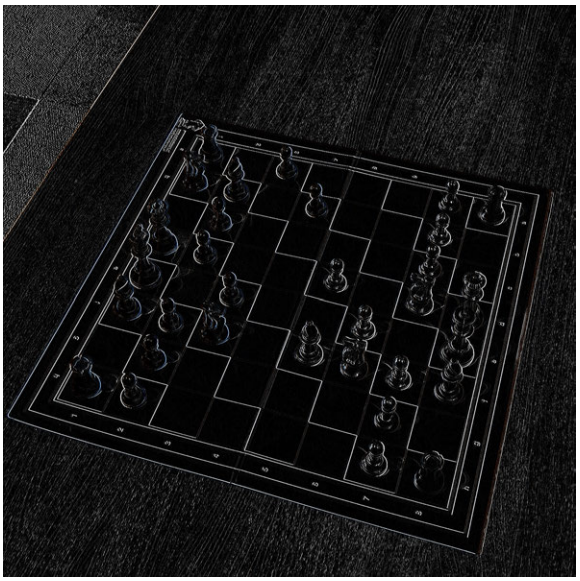
The Prewitt operator is considered to be less sensitive to noise, compared to the Sobel-Feldman operator, but also slightly less accurate. Figure 3.2 illustrates the results of applying the Prewitt operator

on an image sample of ChessReD, with minimal visual differences to the results obtained by the Sobel-Feldman filter in Figure 3.1.

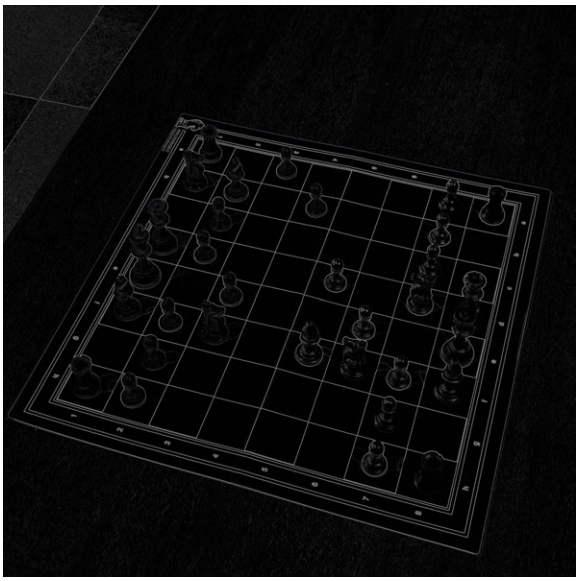
(a) Original image



(b) Horizontal filter



(c) Vertical filter



(d) Magnitude

Figure 3.2: Results of applying the Prewitt filter on an image sample of ChessReD

3.1.3. Canny edge detector

The Canny edge detector, introduced in 1986 by John Canny, remains even today one of the most widely used algorithms for edge detection. It is a multi-stage algorithm known for its robustness and its ability to provide accurate edge detections while minimizing false positives and noise. Its first step is to reduce the noise in the image using a Gaussian smoothing filter. Then having calculated the magnitude of each pixel (x, y) in the second stage, using a similar approach to the Sobel-Feldman and Prewitt operators, the third stage is *non-maximum suppression*. The final and fourth stage of the algorithm is *hysteresis thresholding*.

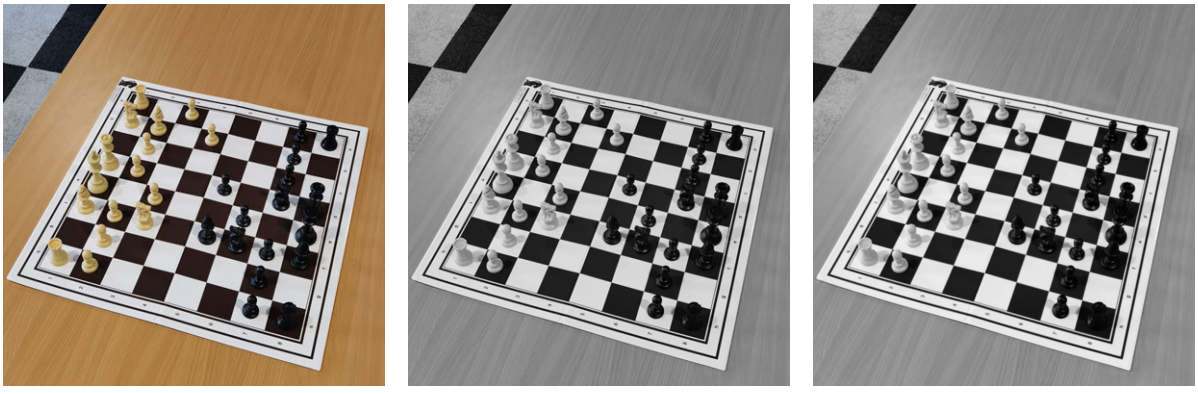
1. **Gaussian smoothing:** Recognizing that convolving the masks directly with the original image for gradient calculation may lead to noise amplification, the first step of the Canny edge detection

algorithm is to apply a Gaussian smoothing filter on the image. Thus, it convolves the image with a Gaussian kernel to both reduce the noise and make the edges more pronounced.

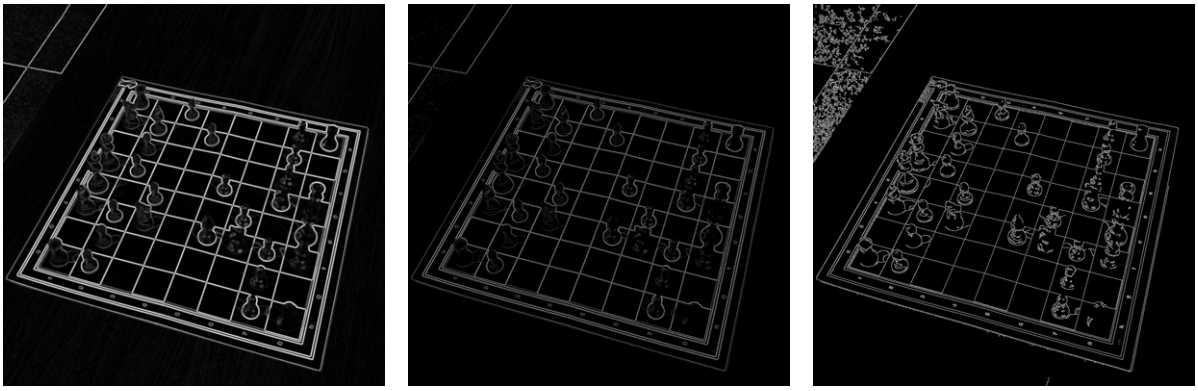
2. **Gradient calculation:** Similar to the Sobel-Feldman and Prewitt edge detection methods, the Canny edge detector uses two 3x3 masks to compute the horizontal and vertical gradients of the image. The weights of the masks are those introduced in the Sobel-Feldman operator (presented in Equation (3.1)).

3. **Non-maximum suppression:** For each pixel, the algorithm compares the gradient magnitude of the current pixel to that of the neighboring pixels, in the direction perpendicular to the edge. If the current pixel's magnitude is greater than its neighbors, the pixel remains as part of the edge. Otherwise, it is suppressed.

4. **Hysteresis thresholding:** In this process, the Canny edge detector uses double thresholding to classify the detected edges into three categories, *strong edges*, *weak edges*, and *suppressed*. Pixels with a gradient magnitude greater than a defined higher threshold ($thresh_H$) are classified as strong edges, while those between the $thresh_H$ and the lower threshold ($thresh_L$) are classified as weak edges. Finally, those with a gradient magnitude less than the $thresh_L$ are suppressed. Subsequently, edge tracking by hysteresis is performed, with the algorithm following the direction of the gradient of the strong edge pixels, and connecting neighboring weak pixels until there are no more weak edges to be linked. The remaining weak edges that are not linked get suppressed, while the linked ones represent the final detected edges.



(a) Original image **(b)** Grayscale **(c)** Gaussian smoothing



(d) Sobel filter **(e)** Final Canny edges **(f)** Inadequate thresholds

Figure 3.3: Results of applying the Canny edge detector on an image sample of ChessReD

Figure 3.3 illustrates the stage-by-stage results of applying the Canny edge detector on an image sample of the ChessReD. While the intermediate conversion of the image to grayscale (Figure 3.3b) is not required, it usually leads to better results. Additionally, by applying the hysteresis thresholding, the Canny algorithm produces thin, well-connected edges, in contrast with the thick one-pixel wide edges

of the Sobel-Feldman filter (Figure 3.3e and Figure 3.3d respectively). Furthermore, in Figure 3.3f we can see the effect of using inadequate thresholds for the hysteresis thresholding process.

3.2. Line detection

Line detection is a fundamental technique in image processing that involves the identification of straight lines present in an image. A line can be defined as a set of connected pixels that share common characteristics (*e.g.* intensity). Essentially formed by detected edges, lines can provide information about the boundaries between objects and highlight their structure. Thus, lines are important features necessary for higher-level computer vision tasks (*e.g.* object detection). There are several different algorithms developed for line detection. In this section, we will discuss some of the most commonly used ones.

3.2.1. Hough transform

The Hough transform (Hough 1962) is a well-known and widely used technique for line detection (Duda and Hart 1972). It represents lines in polar coordinate space and evokes a voting procedure for plausible line locations. In particular, the image is transformed into the Hough domain, where every edge point (x, y) of the image is converted to polar coordinates (ρ, θ). Then, a 2D array is created, called Hough accumulator, that will store the number of votes for a specific (ρ, θ) pair, representing a unique line (Figure 3.4). The size of the accumulator is determined by the minimum and maximum possible values for ρ and θ. Subsequently, the Hough transform loops through all the pixels in the image and if the pixel lies on a line, a vote is cast for that particular line in the Hough accumulator. Finally, the algorithm searches for the peaks (*i.e.* high number of votes) in the accumulator to define lines.

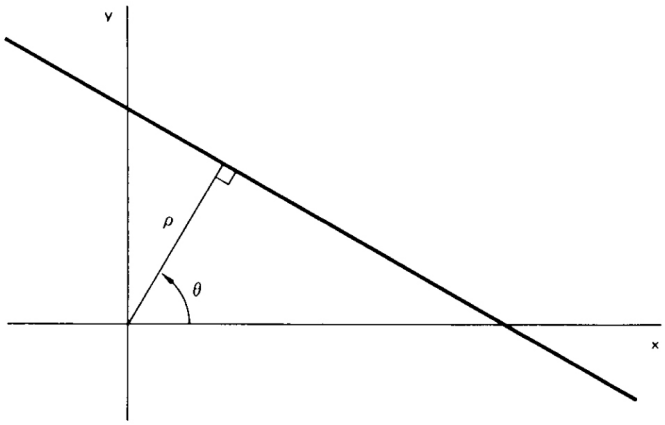
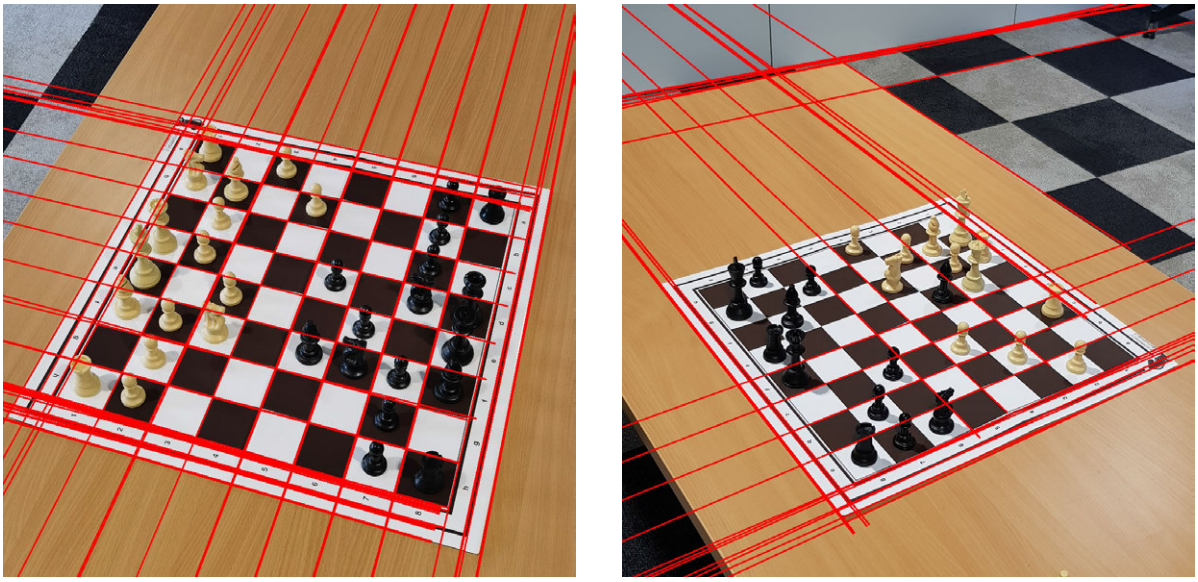


Figure 3.4: Parametric description of a line (source: Duda and Hart 1972)

Figure 3.5 illustrates two examples of detected lines by the Hough transform algorithm, using the same ρ and θ parameters. In Figure 3.5a, we can see that all of the chessboard's lines have been detected, whereas in Figure 3.5b several lines are missing.

Several optimizations of the original Hough algorithm have been proposed. Kiryati, Eldar, and Bruckstein 1991 introduced Probabilistic Hough Transform (PHT), which is considered to be more efficient than the original algorithm, especially in cases where the majority of the pixels in the image are not part of a line. The main difference is that in PHT not all edge points are considered perspective lines and therefore put to voting. It conducts a random point sampling, reducing substantially the number of points to be processed. Furthermore, Matas, Galambos, and Kittler 2000 proposed an even more efficient algorithm, the Progressive Probabilistic Hough Transform (PPHT). In this method, the Hough space is divided into a number of bins and instead of voting for lines at each pixel, a vote is cast for the bin that the pixel lies in. Then, the probability of a line is based on the number of votes of the bin it belongs to. This way, the most salient features of the image are likely to be detected first.



(a) Successful line detection

(b) Inadequate line detection

Figure 3.5: Results of applying the Hough transform on two image samples of ChessReD

3.2.2. Random Sample Consensus (RANSAC)

The Random Sample Consensus (RANSAC), introduced by Fischler and Bolles 1981, is a robust algorithm used to fit models to data with outliers. In the case of line detection, it can effectively discard outlier edges and image noise, which can significantly affect the accuracy of the method. It iteratively estimates line model parameters from randomly sampled edge points until they fit the majority of the data. The RANSAC algorithm consists of the following steps:

1. **Random sample selection:** Randomly select two edge points to uniquely define a line.
2. **Parameter estimation:** Estimate the line parameters (slope and y-intercept) based on the randomly selected points.
3. **Inlier identification:** Determine the number of edge points that fall into this line (*inliers*), with a user-defined tolerance.
4. **Line validation:** If the number of inliers is greater than a pre-defined threshold, proceed to the next step. Otherwise, repeat the first three steps.
5. **Line selection:** After a pre-define number of iterations, or when a specific number of inliers have been found, select the lines with the largest number of inliers.

While the RANSAC algorithm is able to ignore outliers, which makes it robust to noise, it can be computationally expensive for large datasets and it is significantly sensitive to the predefined thresholds for the tolerance and the number of line inliers.

3.2.3. Least squares fitting

Least squares fitting is a simple, yet effective line detection algorithm. Its goal is to minimize the sum of the squared distances between edge points and a line. The algorithm starts by selecting a number of edge points (x, y), which will be used to calculate the line parameters (*i.e.* the slope and the y-intercept) that minimize the sum of the squared distances between those points and the line. With the sum of the squared distances as the objective, one can use a variety of methods to minimize it, such as gradient descent. The process is repeated until the sum is minimized, and thus a line is defined.

3.3. Line clustering

Once lines have been detected in an image, the next step is to cluster them into groups based on a similarity metric. This is an important processing step for computer vision applications since it provides insights into the underline structure that can be valuable for subsequent analysis. For instance, in chessboard detection, this image processing technique can be employed to cluster chessboard lines into vertical and horizontal groups. In this section, we will discuss common line clustering techniques, such as k-means (MacQueen et al. 1967; Lloyd 1982), hierarchical (Lance and W. T. Williams 1967), and density-based (Ester et al. 1996) clustering.

3.3.1. k-means clustering

The k-means is a widely used unsupervised clustering algorithm that aims to group data points into k clusters. To accomplish that, each data point is assigned to the cluster with the nearest mean value, based on a predefined metric. The step-by-step algorithm is described below.

1. **Initialization:** k initial cluster centroids can be assigned either randomly, or by selecting k random data points.

2. **Assignment:** Each data point is assigned to the cluster whose centroid is nearest based on a predefined distance metric (*e.g.* Euclidean distance)

3. **Update:** The centroids of the clusters are re-positioned to the mean values of their members.

4. **Iteration:** Steps 2 and 3 are repeated until convergence (*i.e.* no changes for a number of iterations) or a predefined number of iterations is reached.

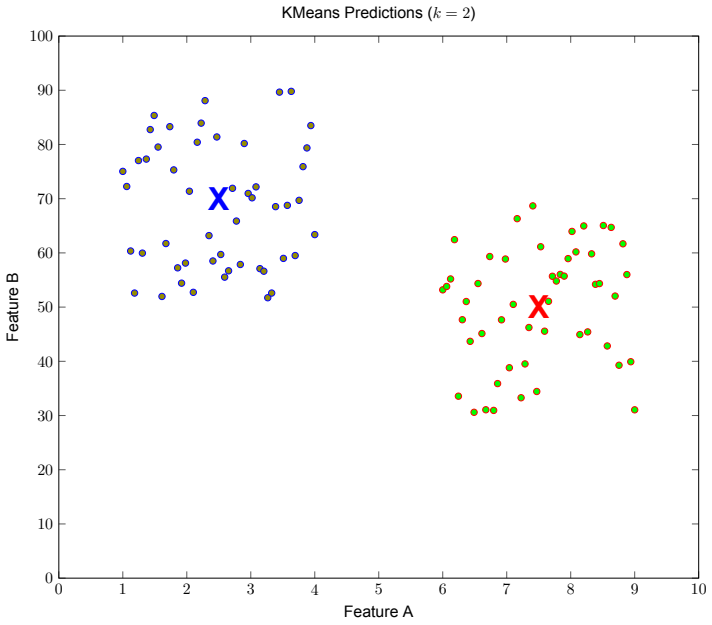


Figure 3.6: Example of a converged k-means clustering with k=2

In the case of line clustering, the first step would be to identify the characteristic based on which the lines would be clustered (*e.g.* orientation, length, etc). For chessboard detection, a common case is to cluster them based on orientation (*i.e.* horizontal and vertical chessboard lines). Then, the data points would be the different slopes of the lines and the number of clusters (k) would be 2.

3.3.2. Hierarchical clustering

Hierarchical clustering is a clustering method that builds a tree-like structure of the clusters (Figure 3.7), generating a hierarchical representation of the data points. Depending on the hierarchical clustering type, agglomerative (bottom-up) or divisive (top-down), clusters are merged or split, respectively, based

on their similarity. The different steps to execute the agglomerative hierarchical clustering are presented below.

1. **Initialization:** Start with each line as its own cluster.
2. **Pairwise distance calculation:** Calculate the pairwise distance between all clusters.
3. **Merging:** Merge the two "closest" clusters based on the calculated distances into a single cluster.
4. **Iteration:** Repeat steps 2 and 3 until a stopping criterion is met (*e.g*. target number of remaining clusters)

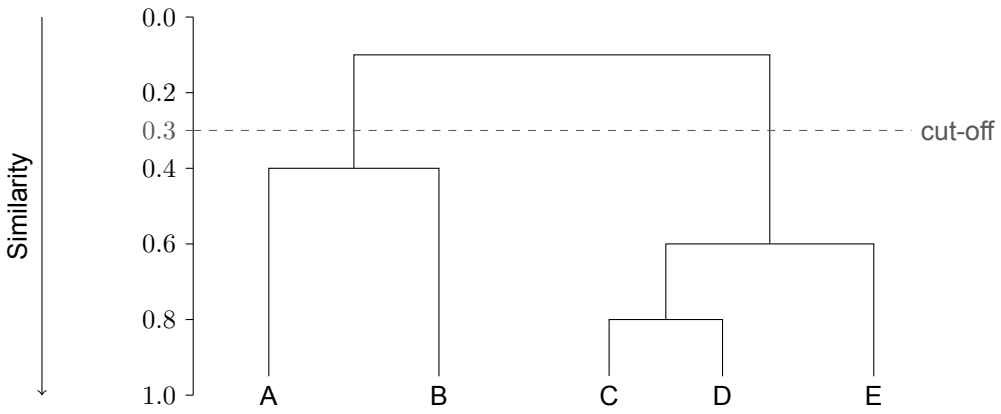


Figure 3.7: Example of an agglomerative hierarchical clustering with cut-off at 2 clusters

The way that the "closest" clusters are computed at step 3, defines the type of *linkage* for the algorithm. For *single linkage*, the minimum distance between the two clusters is computed, whereas for *complete linkage*, the maximum distance between two clusters is computed. Other types include the *average linkage*, which computes the average distance between clusters, and *centroid linkage*, which computes the distances between the calculated centroids for each cluster.

In the context of chessboard detection, agglomerative hierarchical clustering can be used to create two top-level groups (*i.e*. horizontal and vertical) based on the lines' orientations. The data points, again, would be the different slopes of the lines, and the algorithm would repeat merging clusters until only two were left. Figure 3.8 illustrates an example of this approach where detected lines have been clustered to horizontal and vertical for an image of ChessReD, using single linkage.

3.3.3. Density-based clustering

Density-based clustering methods aim to identify clusters based on the density of the data points in the feature space. Data points in high-density areas are clustered together, while those in low-density areas are considered noise or border points. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)(Ester et al. 1996) is one of the most popular density-based clustering algorithms. In DBSCAN, data points are separated into three categories, namely *core points*, *border points*, and *noise points*. Core points are the data points that are surrounded by a predefined number of other points, within a predefined ε-neighborhood area. Border points are those that are not core points, but are within the ε-neighborhood of a core point. The rest of the points are considered noise. Thus, DBSCAN requires the following two parameters to be specified:

- **Epsilon (ε):** Defines the neighborhood and represents the maximum distance between two points for them to be considered neighbors.
- **Minimum points:** The minimum number of points required within the ε-neighborhood for a point to be considered core.

In practice, the algorithm works as follows.

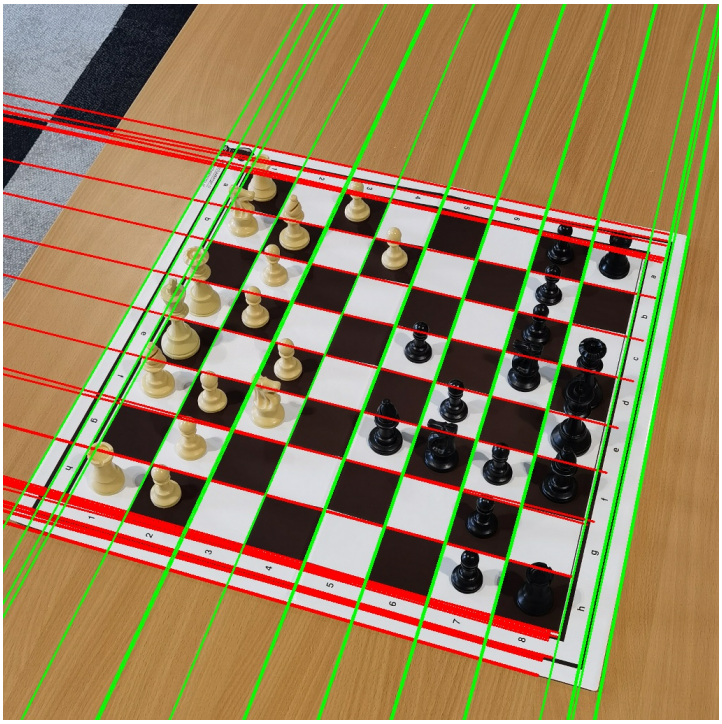


Figure 3.8: Results of applying agglomerative hierarchical clustering on the slopes of the detected lines of Figure 3.5a for an image sample of ChessReD

1. **Initialization:** The algorithm randomly selects a data point and determines its ε-neighborhood area.

2. **Cluster formation:** If the selected node is a core point, a cluster is formed and it includes all the *density-reachable* points from the core point. A point is considered density-reachable from a core point when it is within its ε-neighborhood, or when it is within the ε-neighborhood area of a core point that can be reached through a chain of core points that are density-reachable.

3. **Noise points:** If the selected point was not a core, it is marked as visited and noise. An initially marked noise point can be converted to border point in a following iteration.

4. **Iteration:** If the cluster formation is finished (*i.e.* all density-reachable points have been included in the cluster), or if the previously selected point was a noise one, the algorithm selects a new unvisited data point at random and the whole process is repeated.

The DBSCAN algorithm leverages density information to automatically identify clusters of varying shapes and sizes. The number of clusters does not have to be predefined, as it is inferred by the density of the data points. In chessboard detection, this algorithm can be used to cluster similar lines together based on orientation or distance from specific points, among others.

3.4. Homography matrices

In the scope of 2D computer vision, a homography matrix is a 3x3 matrix, usually denoted as H, that represents a projective transformation. Projective is a transformation that maps points from one plane to another. Thus, in the case of an image, a (x, y) point in the original image will be mapped to the transformed coordinates (x', y') of the target image. To compute the homography matrix, at least four pairs of corresponding points from the two images are required. In chessboard detection, these four points could correspond to the corners of the chessboard and the homography matrix can be used to transform the image of the chessboard from the given viewing angle to top-view. There are several methods that can be used to compute the homography matrix, but two of the most commonly used are

the Direct Linear Transform (DLT)(Dubrofsky 2009) and the Random Sample Consensus (RANSAC) (Fischler and Bolles 1981).

3.4.1. Direct Linear Transform (DLT)

The DLT algorithm is used to compute the homography matrix for projective transformations between two images. It aims to minimize the error between the projected points in the target image and the corresponding points in the original one, while each pair of points represents the same point from a different perspective. The algorithm can be summarized as follows.

1. **Find correspondences:** The algorithm requires at least four corresponding pairs of points between the target image and the original one, so the first step is to find those pairs.

2. **Form the homogeneous linear system:** From the projective transformation equations, a homogeneous linear system can be derived for the four correspondence pairs, resulting in a linear system of 2N equations, with N being the number of correspondence pairs.

3. **Solve the linear system:** The linear system can be solved using various techniques, such as Singular Value Decomposition (SVD).

4. **Compute homography matrix:** The solution vector h of the linear system represents the elements of the Homography matrix H. Thus, reshaping the solution vector to a 3x3 matrix produces the homography matrix.

3.4.2. Random Sample Consensus (RANSAC)

The RANSAC algorithm presented in Section 3.2.2 can also be used to estimate the homography matrix. It iteratively selects a random subset of point correspondences and fits a homography model to them. Then, it computes the number of inlier-points that are consistent with the estimated model. An overview of the method is presented below.

1. **Random sample selection:** In each iteration, a minimal random sample of four point correspondences is selected.

2. **Candidate homography matrix computation:** The candidate homography matrix is estimated using a computation method (*e.g.* DLT) based on the selected correspondence pairs.

3. **Inlier selection:** The algorithm computes the inlier points, which are the correspondence points within a predefined distance threshold of the projected points.

4. **Model update:** The homography model gets updated to the one with the largest number of inliers.

5. **Iteration:** The above steps are repeated until a termination criterion has been met (*e.g.* number of iterations or target number of inliers)

6. **Final homography matrix computation:** Once the loop is terminated, the homography matrix is computed based on the inliers of the obtained model.

In the scope of chessboard detection, the RANSAC algorithm could be used on the intersection points of the detected horizontal and vertical lines (*i.e.* square corners). Ideally, the homography model that would include the largest number of inliers, would correspond to the chessboard's corners, and thus the computed homography matrix would optimally project the chessboard, correcting perspective distortions. Applying this process to the detected horizontal and vertical lines of the example in Figure 3.8, we can obtain the results of Figure 3.9.

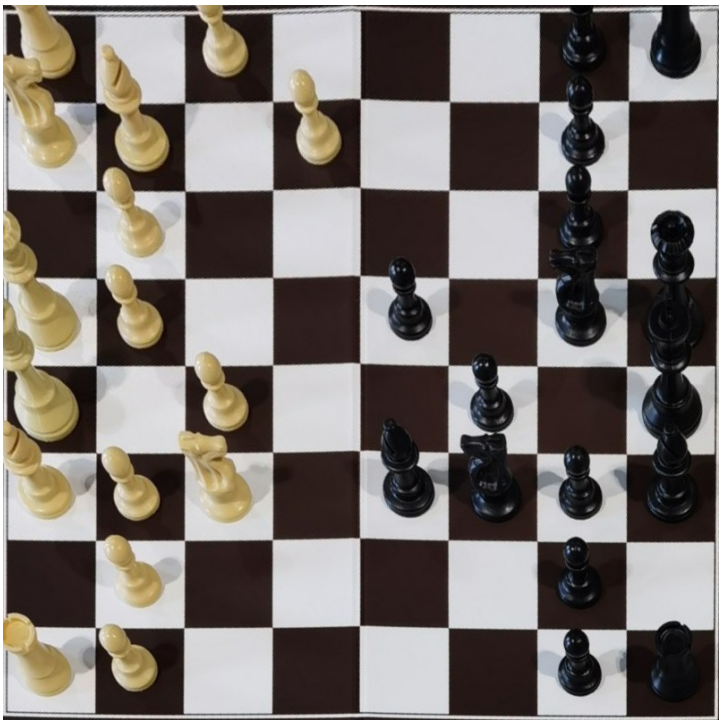


Figure 3.9: Results of applying the RANSAC-based homography estimation method on the detected horizontal and vertical lines of Figure 3.8 for an image sample of ChessReD

4

Deep Learning

Deep learning is a type of machine learning, inspired by the human brain, that utilizes *Artificial Neural Networks (ANNs)* to learn complex patterns in data. While its foundation dates back to the 1940s when the first neuron model was created by McCulloch and Pitts 1943, significant advancements in neural networks did not happen until the introduction of the *perceptron* architecture (Rosenblatt 1958), the *Stochastic Gradient Descent (SGD)* (Robbins and Monro 1951) algorithm, and the backpropagation algorithm (Rumelhart, Hinton, and R. J. Williams 1986), which enabled the training of *multi-layer perceptrons (MLPs)*. Since then, a lot of breakthroughs have occurred in the field with improvements in *Deep Neural Network (DNN)* architectures and the introduction of different types of architectures, from *Convolutional Neural Networks (CNNs)* (Fukushima 1980; LeCun et al. 1989) and *Recurrent Neural Networks (RNNs)* (Hopfield 1982; Elman 1990), all the way to *Transformers* (Vaswani et al. 2017).

This chapter is organized as follows. Section 4.1 presents an overview of neural networks, along with activation functions, loss functions, and details about the training process. In Section 4.2, we will delve into a specific type of neural networks, the Convolutional Neural Networks (CNNs). Then, Section 4.3, includes a brief introduction of the Transformers architecture and its different variants for computer vision tasks.

4.1. Neural Networks

Neural networks, a fundamental component of deep learning, enable machines to learn complex patterns in data. Inspired by the structure of the human brain, neural networks consist of artificial neurons, organized into layers, and the connections between them.

4.1.1. Multi-layer Perceptrons (MLPs)

A multi-layer perceptron (MLP), also known as feedforward neural network, is the simplest form of neural network. It is composed of multiple layers of interconnected artificial neurons. The first layer (*i.e. input layer*), receives the raw data, then one or more *hidden layers* are used to extract features and transform the output of the previous layer into a new representation of the data. Finally, the *output layer* produces the model's predictions or classifications. Furthermore, in a multi-layer perceptron, each of the neurons is connected to all of the neurons of its previous and subsequent layers (*i.e. fully-connected*). An overview of the MLP architecture is illustrated in Figure 4.1.

The alternative name, feedforward neural network, was given to MLPs because of the way that the final output is computed. Layer by layer the output of a neuron is forwarded as input to the neurons of the subsequent layer. More specifically, the input layer receives the raw data, typically in the form of a feature vector, and passes it to the neurons of the first hidden layer. The output of a neuron in a hidden layer is determined by the weighted sum of the inputs of the previous layer as in Equation (4.1).

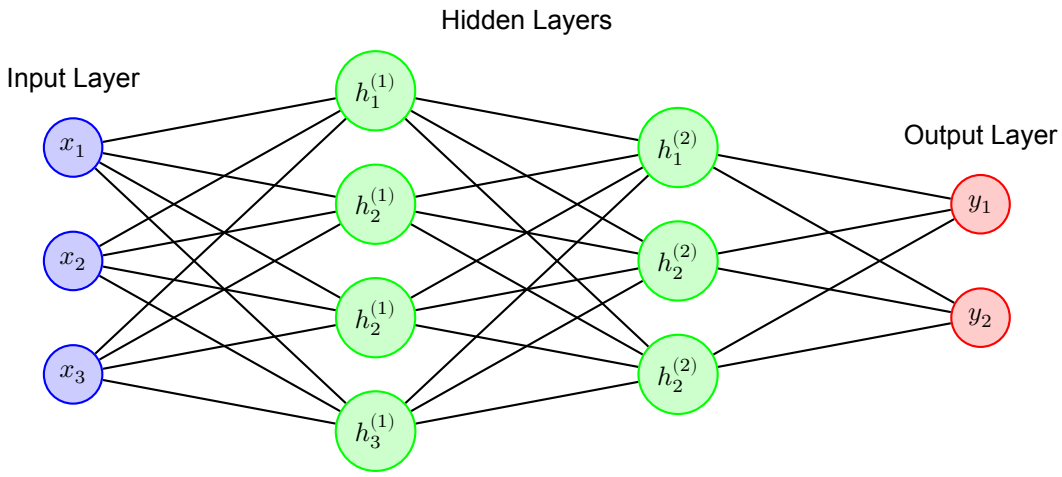


Figure 4.1: Multi-layer Perceptron with 2 hidden layers

$$z_j = \sum_{i=1}^{n} (w_{ij} \cdot x_i) + b_j \tag{4.1}$$

Where:

- z_j is the weighted sum for neuron j of the current layer
- n is the number of neurons in the previous layer
- w_{ij} is the weight of the connection between neuron i of the previous layer to neuron j of the current layer.
- x_i is the output of neuron i of the previous layer
- b_j is a bias term for neuron j of the current layer

Subsequently, the weighted sum is usually passed through an *activation function* f to compute the output $y_j = f(z_j)$ of neuron j. Activation functions, further discussed in Section 4.1.3, are used to introduce non-linearity to the network, and thus allow it to learn more complex data representations. Then, the output layer performs a similar computation for the weighted sum, but the activation function to be used depends on the task that the network has to perform.

4.1.2. Deep Neural Networks (DNNs)

A deep neural network (DNN) is a type of MLP that has several hidden layers. The introduced depth has enabled DNNs to learn more complex patterns in data and create hierarchical representations. The number of hidden layers can vary, but typically there are at least two hidden layers. Increasing the networks depth can facilitate the learning of more complex representations, but with greater computational cost. Figure 4.2 shows an example of a deep neural network architecture with n hidden layers.

DNNs have shown remarkable success, achieving state-of-the-art results, in various domains, including computer vision, natural language processing, and speech recognition. The large availability of data and computing resources, in combination with improved architectures and training techniques, have contributed to the success of DNNs.

4.1.3. Activation functions

Activation functions are used in neural networks to introduce non-linearity, which in turn allows the network to learn more complex patterns in data. They are differentiable operators, usually with no

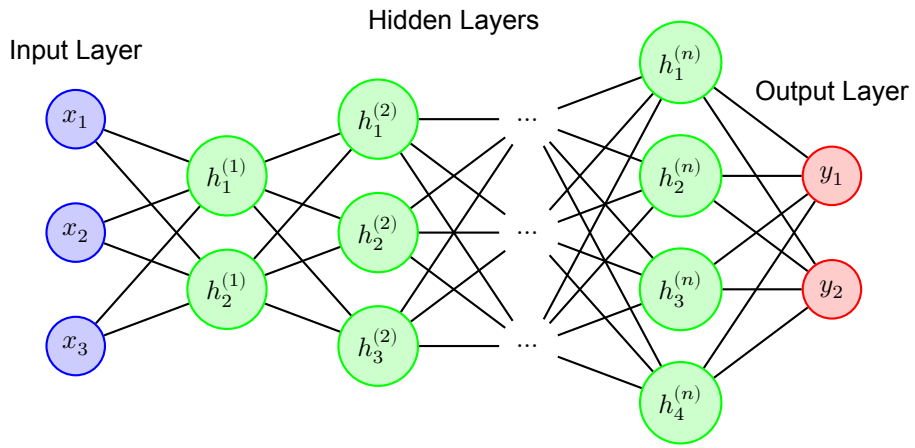


Figure 4.2: Deep Neural Network architecture with n hidden layers

training parameters. Because of the significance of their role, several activation functions have been researched. In this section, we will focus on some of the most commonly used.

The **Sigmoid** activation function, also known as the *logistic* activation function, was introduced by Rumelhart, Hinton, and R. J. Williams 1986 in their search for a differentiable, non-linear activation function. It maps its input to a range between 0 and 1 based on the formula of Equation (4.2).

$$sigmoid(z) = \frac{1}{1 + e^{-z}} \tag{4.2}$$

Where z is the weighted sum calculated at the neuron. A similar activation function that is commonly used is the **Hyperbolic Tangent (Tanh)** function (Equation (4.3)). Their difference is that while the sigmoid function maps the input between 0 and 1, the tanh maps it between -1 and 1. This wider range of output values constitutes this function useful when the data is centered around zero.

$$tanh(z) = \frac{2}{1 + e^{-2z}} - 1 \tag{4.3}$$

A more computationally efficient, and popular, activation function is the **Rectified Linear Unit (ReLU)** (Fukushima 1975), presented in Equation (4.4). ReLU is a non-linear function with linear behavior for positive inputs (*i.e.* remain unchanged) and zero output for negative inputs. It is a widely used activation function because other than its simplicity and computational efficiency, it also addresses a common issue of the sigmoid and tanh functions, the *vanishing gradients* (Hochreiter 1998).

$$relu(z) = max(0, z) \tag{4.4}$$

Variations of the ReLU activation function have also been proposed. The **Parametric ReLU** (He et al. 2015) function (Equation (4.5)) introduces a trainable coefficient that allows leakage of gradients for negative inputs, preventing neurons from getting stuck as deactivated during training.

$$prelu(z) = \begin{cases} z & \text{, if z>0} \\ \alpha \cdot z & \text{, otherwise} \end{cases} \tag{4.5}$$

Where α is the trainable coefficient for the negative inputs. For $\alpha = 0.01$, the activation function is also known as *Leaky ReLU*. Finally, the **Softmax** activation function, primarily used at the output layer of

neural networks trained for multi-class classification tasks, converts the vector of real scores produced by the output layer to class probabilities that sum to 1. The formula is presented in Equation (4.6).

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}} \tag{4.6}$$

Where C is the number of classes in the classification task. Figure 4.3 illustrates the aforementioned activation functions, with the exception of softmax because its output represents a probability distribution and we cannot visualize multiple values simultaneously. Also, for visualization purposes in Figure 4.3d we have set the α coefficient of the Parametric ReLU function to 0.3, while in practice α is a trainable parameter.

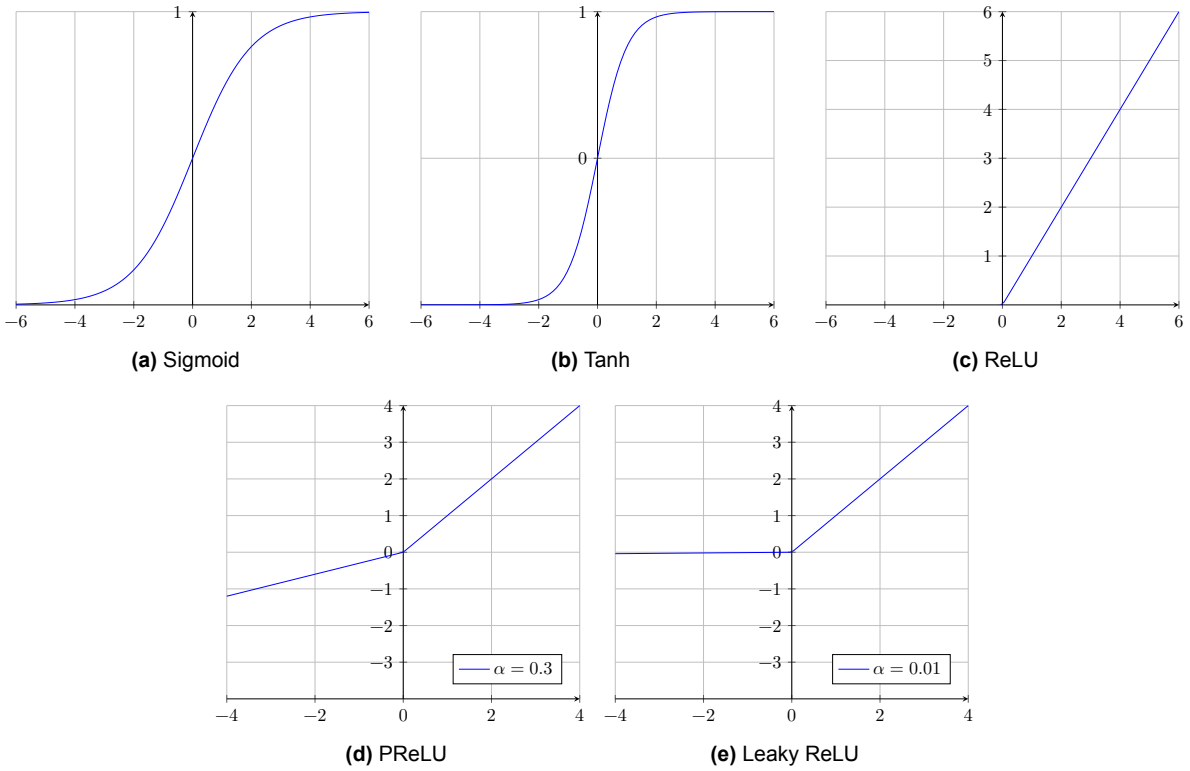


Figure 4.3: Common activation functions

4.1.4. Loss functions

Loss, or cost, functions are significant components of neural network training. They are used to measure the error between the predicted output and the actual, expected output (ground truth). The calculated loss is then used to update the weights of the neural network during training. The choice of an appropriate loss function depends on the nature of the task (*e.g.* regression, classification, etc.). Thus, several different loss functions that can be used. In this section, we will present some of the most commonly used ones.

The **Mean Squared Error (MSE)** (Lehmann and Casella 2006) is a loss function that measures the average squared difference between the predicted and the actual outputs. It is widely used for regression tasks, where the models are trained to predict continuous values. Equation (4.7) shows how to compute the MSE loss for a dataset with N data samples.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.7}$$

Where y_i is the actual output for sample i and $\hat{y}_i$ is the corresponding predicted output.

For classification tasks where the goal is to predict discrete values, the **Cross-Entropy** loss function, or *log loss*, is commonly used when training neural networks. Initially introduced by Shannon 1948, the cross-entropy is a measure of the difference between the predicted class probabilities and the actual ones. Equation (4.8) shows how to compute the cross-entropy loss for a classification dataset with N data samples and C classes.

$$CrossEntropy = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} log(\hat{y_{ij}}) \tag{4.8}$$

Where y_{ij} is the one-hot encoded actual output for sample i and class j, and $\hat{y_{ij}}$ is the corresponding predicted probability.

4.1.5. Optimization

The term optimization, in deep neural networks, refers to the process of finding an optimal set of model parameters that minimize the selected loss function. Optimization algorithms are used during training to iteratively update those parameters to minimize the loss on a training dataset. A widely known optimization algorithm for training DNNs is **Gradient Descent** (Cauchy et al. 1847). It is an optimization algorithm that iteratively updates the parameters of the model in the direction of the steepest descent (*i.e.* the negative gradient of the loss function). In particular, for each parameter θ_i of the model, the update is given by the formula of Equation (4.9).

$$\theta_i' = \theta_i - \eta \cdot \frac{\partial L}{\partial \theta_i} \tag{4.9}$$

Where η is a chosen learning rate and $\frac{\partial L}{\partial \theta_i}$ is the partial derivative, or *gradient*, of the loss function L with respect to parameter θ_i for all training samples. Thus, for each gradient descent step, the algorithm has to compute the gradient for the entire training dataset. However, deep neural networks are usually trained on large datasets with millions of samples, constituting gradient descent inefficient.

A variant of gradient descent that addresses the aforementioned issue is **Stochastic Gradient Descent (SGD)** (Robbins and Monro 1951; Kiefer and Wolfowitz 1952). In contrast to regular gradient descent, SGD uses a single training sample at each step to update the model's parameters. While this solution is computationally efficient, it can cause the network to overfit on the training data. Thus, a variant of SGD, called **mini-batch stochastic gradient descent**, is widely used to train neural networks. The term "mini-batch" refers to a small subset of the training samples that are used at each step to update the network's parameters. This technique can improve the model's ability to generalize, without sacrificing computational efficiency.

4.1.6. Backpropagation

In the previous section, it was mentioned that in order to find the optimal network parameters that would minimize its loss function, the gradient of the loss function with respect to the network's parameters needs to be computed. **Backpropagation**, proposed by Rumelhart, Hinton, and R. J. Williams 1986, is an efficient technique to calculate those gradients. It consists of two steps, the *forward propagation* and the *backward propagation*. In the forward propagation, or forward pass, the input data are processed layer by layer until the created intermediate feature representations reach the output layer, which predicts the output. Subsequently, the backward propagation, or backward pass, starts with the calculation of the gradient of the loss function with respect to the model's output. Then, the computed gradient is backpropagated through a recursive algorithm. The algorithm starts from the output layer and recursively goes backwards, layer by layer, using the chain rule of calculus to calculate the derivatives, until it reaches the input layer. When all of the gradients have been computed, the network's parameters can be updated as described in the previous section.

4.2. Convolutional Neural Network (CNNs)

Convolutional Neural Networks (CNNs) (LeCun et al. 1989) are a specific type of neural network suitable for processing data with a grid-like structure. They were inspired by the work of Fukushima 1980, which introduced the idea of complex cells that would hierarchically aggregate information, from simple features into more complex ones. Similarly, CNNs employ multi-layered architectures to progressively learn complex features. Due to their unique characteristics, CNNs are particularly effective in a variety of image-related tasks, exploiting the grid-like structure of images.

In a convolutional neural network, the layers at the lower levels of the architecture (*i.e.* closer to the input) focus on learning low-level features like edges and texture, while the higher-level layers learn abstract features like shapes of objects. This feature extraction and aggregation is a result of the main building blocks of such networks. Traditionally, a convolutional neural network consists of convolutional layers, pooling layers, and fully connected layers. These layers also enable the network to capture local patterns and spatial relationships between objects in images. An overview of a CNN architecture is illustrated in Figure 4.4.

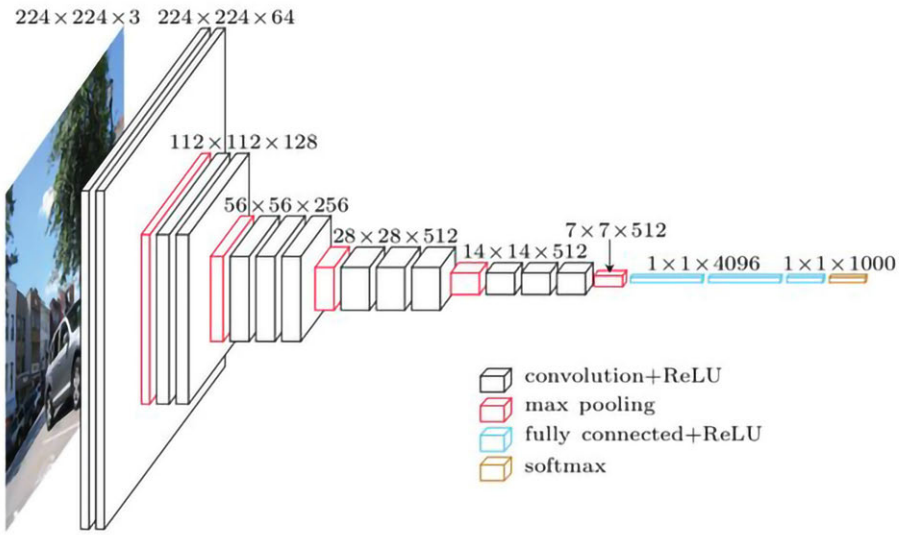


Figure 4.4: Convolutional Neural Network architecture of VGGNet by Simonyan and Zisserman 2014 (source: Bačanin Džakula et al. 2019)

In this section, we will discuss the main building blocks of convolutional architectures. Additionally, some more recent advancements, **Residual Networks (ResNets)** (He et al. 2016), that allowed CNN architectures to go deeper and substantially improve their performance will be presented. Finally, we will discuss the variation of ResNet, named **ResNeXt** (Xie et al. 2017), that was used in the experiments of Chapter 2.

4.2.1. Convolutional layers

Convolutional layers are the foundation blocks of CNNs. They are responsible for extracting meaningful features from input images, relying on their core operation, **convolution**. The convolution is a mathematical operation between two variables, a *kernel* and a *feature map*. The kernel, also known as filter, is a small matrix of learnable weights that is slid over the feature map, which is data in a grid-like format. While the kernel is slid over the feature map, the dot product between them is computed at each location. The output of the convolution is a new feature map that contains the extracted features from the operation. Mathematically, the convolution for 2-dimensional inputs can be defined as follows (Equation (4.10)).

$$F'(i,j) = (K * F)(i,j) = \sum_m \sum_n K(m,n) F(i+m, j+n) \tag{4.10}$$

Where F' is the output feature map, F is the input feature map, and K is the kernel. While Equation (4.10) in fact describes the *cross-correlation* formula, in the scope of deep learning convolution and cross-correlation are used interchangeably. Figure 4.5 illustrates an example of the convolution operation.

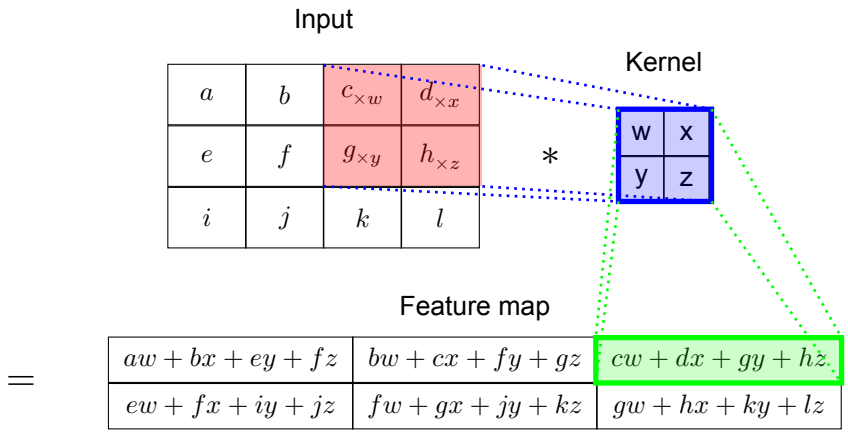


Figure 4.5: Convolution operation with a stride of 1

Because of the *sliding-window* function of the convolution operation, it only considers a small region of the input feature map at a time, allowing it to capture local patterns. The size of this region, with respect to the initial input image, is called *receptive field*. The layer's extracted features of each region are passed to the next convolutional layer as input. The deeper the convolutional block is in the network, the larger its receptive field (*i.e.* hierarchical features).

Finally, a significant property of convolutional layers is *weight sharing*. While there can be several kernels in a convolutional layer, the exact same kernels are slid across the entire input feature map, and thus the same weights are used to extract features from different regions of the input. This property constitutes convolutional layers efficient for computer vision applications where the size of the input images can be substantially large for high resolutions.

4.2.2. Pooling layers

Pooling layers are another important component of convolutional neural networks. Their role is to downsample (*i.e.* reduce spatial dimensions) the output of preceding convolutional layers, enabling the network to learn more high-level features, while also reducing the computational complexity. The pooling operators, similar to the convolution, are applied on an input feature map using a sliding window technique. The distinction is that the cross-correlation operator is replaced by an aggregation function. Based on the selected aggregation function, there are two commonly used types of pooling layers, the **Max Pooling** and the **Average Pooling**. The max pooling operator extracts the maximum value of the input feature map at the corresponding region that it is applied, whereas the average pooling extracts the mean value.

A parameter of the pooling operation that is commonly used is the window *stride*. It determines the step size at which the window slides over the input feature map during the operation. The higher the stride, the greater the downsampling. It should be noted that the stride parameter is not unique to the pooling operation. It can also be used in convolutions resulting in what is called *strided convolutions*. Figure 4.6 illustrates an example of using the pooling operations with a window size of 2 and a stride of 2.

4.2.3. Fully-connected layers

Fully-connected layers, also known as *dense layers*, are usually found usually at the end of a CNN architecture. They allow the network to learn global relationships and high-level features based on the feature map representations obtained by the preceding convolutional layers. Thus, the output of the fully-connected layer is used to make the predictions. Figure 4.7 illustrates an example of fully

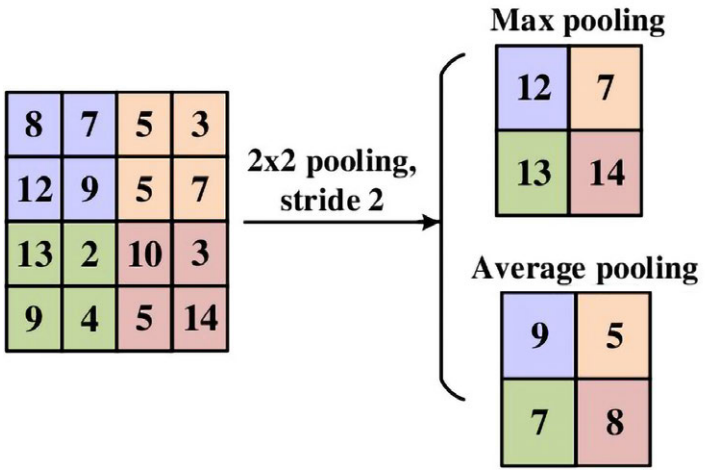


Figure 4.6: Pooling operations (source: Yingge, Ali, and Lee 2020)

connected layers.

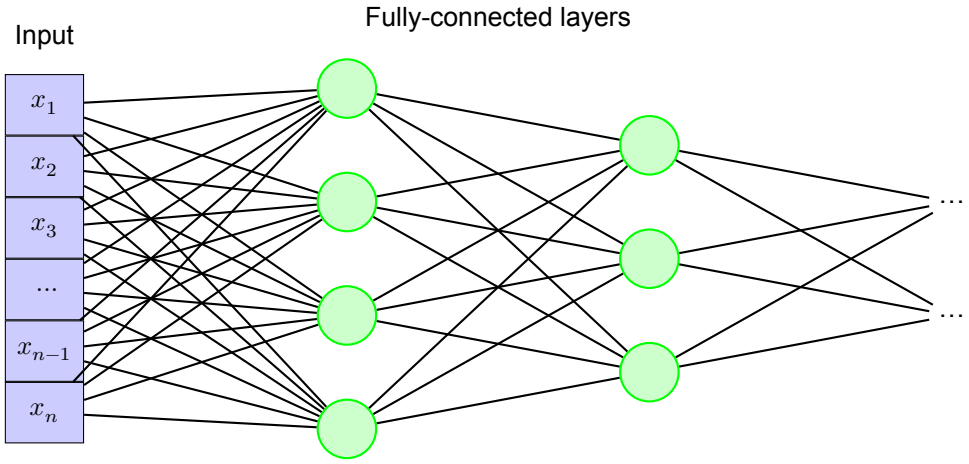


Figure 4.7: Fully-connected layers

Operation-wise, before applying a fully-connected layer, the feature map of the previous layer is flattened. Then, all of the activations of the flattened feature map are connected with each neuron of the fully connected layer. Subsequently, these neurons apply a linear transformation of the inputs, similar to a simple perceptron, using a weight matrix. Then, non-linearity is applied by means of an activation function, which is selected based on the task that the network has to perform. The output size of the fully-connected layer determines the size of the predictions. For instance, in a multi-class classification problem the output size of the last fully-connected layer would equal the number of classes in the task. Next, a softmax activation function would be used to convert the layer's output values into class probabilities.

4.2.4. Residual Networks (ResNets)

When convolutional neural network architectures first started to get deeper, the problem of *vanishing gradients* appeared. In Section 4.1.6, we described the process of backpropagating the loss, after a forward pass, to update the network's parameters during training. Backpropagation includes the calculation of the derivative, or gradient, of the loss function with respect to the network's parameters using the chain rule of calculus. The deeper the network architecture, the longer the chain. Thus, the gradients become extremely small when they backpropagate through multiple layers, with layers in the lower level of the architecture (*i.e.* closer to the input) receiving negligible updates. This phenomenon

is called vanishing gradients.

Residual Networks (ResNets) (He et al. 2016) address this problem using a specific type of building block, the residual blocks. A **residual block** mainly consists of two convolutional layers and a skip connection between the input and the output of the block. The skip connection enables the network to essentially bypass the convolutional layers, thus creating an alternative gradient path to mitigate the problem of vanishing gradients. At the output of the residual block, an element-wise addition is performed between the output of the second convolutional layer and the block's input, as shown in Figure 4.8.

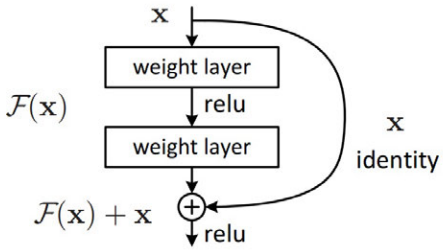


Figure 4.8: Residual Block (source: He et al. 2016)

ResNets are constructed by stacking several residual blocks together. The authors proposed architecture variants with 18, 34, 50, 101, and 152 layers without encountering the vanishing gradients problem. For the deeper architectures, they also introduced the **bottleneck residual blocks** to reduce the computational time. These blocks contain a stack of 3 convolutional layers, instead of 2 in the basic residual block. The first and the last of the layers are 1x1 convolutions to reduce and then restore the dimensions of the block's input, thus creating a bottleneck for the middle 3x3 convolutional layer. This modification (Figure 4.9) leads to a deeper network with similar time complexity to its shallower version.

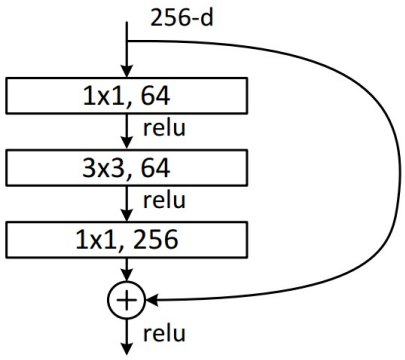


Figure 4.9: Bottleneck Residual Block (source: He et al. 2016)

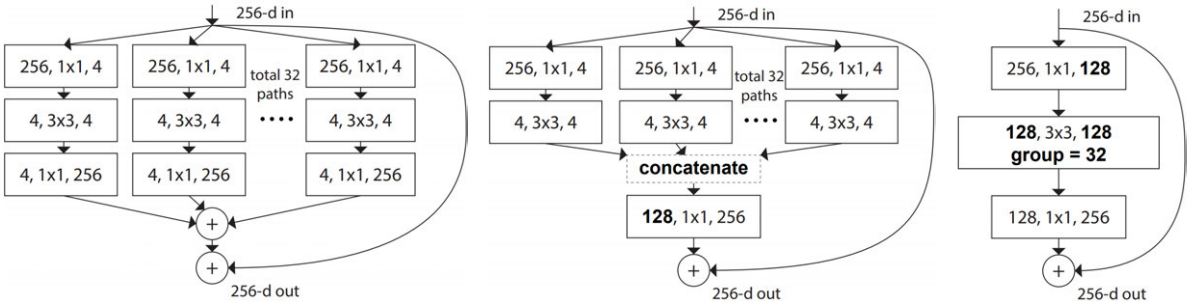
Additionally, the skip connections in the residual blocks helped alleviate another common issue of the deeper architectures, the *degradation problem*. The problem refers to cases where increasing the number of layers in the network, leads to performance degradation, as evidenced by higher training error. However, with the skip connections, also known as identity mappings, ResNets ensure that their deeper architecture variants won't have a training error greater than their shallower counterparts.

Due to the profound impact that ResNets have had in the field of computer vision, they have become a fundamental architectural design that inspired several well-known architectures (Zagoruyko and Komodakis 2016; Huang et al. 2017; Xie et al. 2017). Additionally, ResNets' effectiveness has established them as a preferred backbone network for feature extraction in various computer vision tasks.

4.2.5. ResNeXt

Building upon the success of ResNet, Xie et al. 2017 proposed an architecture with multi-path bottleneck residual blocks, called ResNeXt. The paths within a residual block compute different representations for the same input. Subsequently, these representations are aggregated to form the output of the residual block. This process enhances the network's representational capacity, which in turn leads to better performance. The *"cardinality"* of the network, which refers to the number of parallel paths used within each residual block, is a hyperparameter to be defined along the network's depth (*i.e.* number of layers) and width (*i.e.* number of filters). Additionally, the multiple paths share the same topology in their convolutional layers.

Figure 4.10 illustrates three equivalent building blocks of ResNeXt explored by the authors. In this



(a) Aggregated residual transformations **(b)** Early concatenation **(c)** Grouped convolutions

Figure 4.10: Equivalent building blocks of ResNeXt.The convolutional layers are denoted as (#input channels, filter size, #output channels). (source: Xie et al. 2017)

example the cardinality of the network is 32. The building block of Figure 4.10a performs the aggregated transformation described above. The aggregated output can be computed using the formula of Equation (4.11).

$$y = x + \sum_{i=1}^{C} T_i(x) \tag{4.11}$$

Where y is the output of the block, x is the input, C is the cardinality, and T is the transformation of the path i. Furthermore, it has been proved (Xie et al. 2017) that with some tensor manipulation, Figure 4.10b is equivalent to Figure 4.10a. Finally, in the building block of Figure 4.10c the aggregation is reformulated into grouped convolutions.

Grouped convolution (Krizhevsky, Sutskever, and Hinton 2012) is a variant of the standard convolution operation, where the input channels are divided into a predefined number of groups and the convolution is applied to each group independently. Thus, in the example of Figure 4.10c, the 128 channels at the bottleneck layer are divided into 32 groups of 4 channels each, similar to Figures 4.10a and 4.10b. The individual outputs of the groups are then concatenated within the grouped convolution to form its output. While equivalent, the grouped convolutions were selected out of the three building blocks for ease of implementation.

The aggregation of information obtained from the C different channels in the ResNeXt network, where C is the cardinality, allows the model to learn more complex features, while retaining its computational efficiency. The model's increased capacity enables it to achieve better classification performance than its ResNet counterpart, without additional computational overhead. For this reason, it was selected to be used in the experiments of Chapter 2.

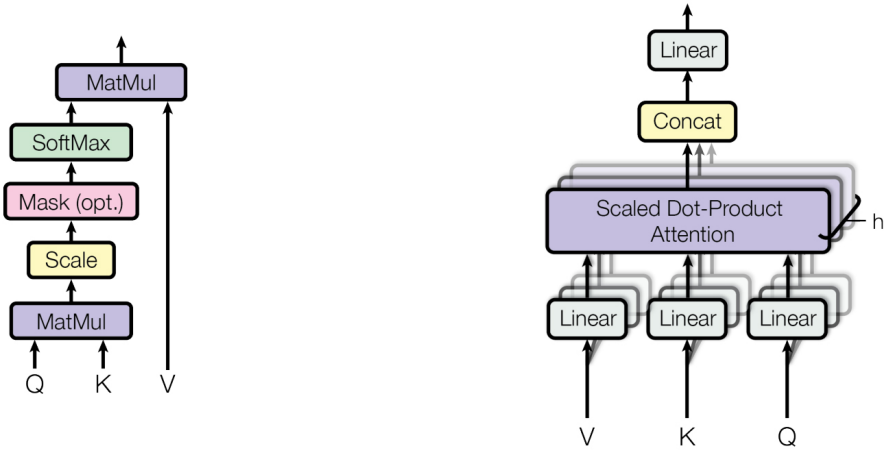
4.3. Transformers

Transformers were originally developed by Vaswani et al. 2017 for natural language processing (NLP) tasks. However, the introduced **self-attention mechanism** which enabled them to attend to different regions of the input, while also capturing long-range dependencies between features, did not go unnoticed in the field of computer vision. Ramachandran et al. 2019 were the first to show that self-attention can replace convolution and achieve state-of-the-art results on vision tasks. Later the same year, Cordonnier, Loukas, and Jaggi 2019 demonstrated that **multi-head self-attention** layers, with a sufficient amount of heads, can express any convolutional layer, thus constituting the latter redundant in their architecture. This finding was thoroughly evaluated by Dosovitskiy et al. 2020, with the authors demonstrating that a pure transformer architecture, the *Vision Transformer (ViT)*, that is applied directly to sequences of image patches can achieve competitive, or even superior, classification performance to that of the then state-of-the-art convolutional models. The same effectiveness has been shown by

transformer variants, typically utilizing **encoder-decoder** architectures, in other computer vision tasks, such as object detection and semantic segmentation.

4.3.1. Self-attention

The self-attention mechanism enables a transformer network to learn the relationships between different pixels in an image. It is computed as a weighted sum of different positions within a transformed pixel sequence. The learnable attention weights determine the relevance of an element of the sequence to the other elements. Additionally, there are three key components of the self-attention mechanism: the *query*, the *key*, and the *value*. The query vector is used to compute the attention scores, or relevance, of an element with respect to all other elements in the sequence. The key vector represents the importance of particular elements in the sequence. Finally, the value vector stores the information of individual pixels in an image.



(a) Scaled dot-product attention

(b) Multi-head attention

Figure 4.11: Attention mechanisms (source: Vaswani et al. 2017)

Figure 4.11a illustrates the self-attention mechanism, called *scale dot-product attention* (Vaswani et al. 2017). To compute its output, the mechanism first computes the attention scores by taking the dot-product between the query and key vectors, and normalizing it by dividing with the square root of their dimension. The dot product measures the similarity between the pairs of elements. Next, a softmax function is applied to the attention scores to compute the attention weights on the values. Finally, a weighted sum of the values is computed as the output of the attention mechanism. Equation (4.12) demonstrates how the output of the attention mechanism is computed.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{4.12}$$

4.3.2. Multi-head attention

Multi-head attention further enhances the model's ability to learn various relationships between different representations of pixels in an image. Using several self-attention layers, the transformer captures different dependencies and representations between sequences of pixels, which are then concatenated to a single representation. This representation is then transformed by a linear layer, which produces the output of the multi-head attention layer (Figure 4.11b). The use of multi-head attention enables the transformer to learn multiple representations for the same sequence of pixels and subsequently use them to capture both local and global features in an image.

4.3.3. Patch and position embeddings

In transformer architectures, multi-head attention is applied to sequences of transformed pixels. Each sequence is constructed by two types of embeddings, *patch* and *position*. To create patch embeddings,

each image is first cropped into smaller fixed-size patches, which are then transformed into 1D embedding vectors, using a learnable linear projection. Similarly, the position embeddings, that introduce information about spatial relationships between patches, are learned by the network during training. Patch and position embeddings are concatenated to produce a sequence that is fed as input to the transformer encoder.

4.3.4. Transformer Encoder

The transformer encoder contains a predefined number of *encoder layers*, which can be adjusted based on the complexity of the task. Each encoder layer has a multi-head self-attention module and a *feed-forward network (FFN)*. The FFN typically consists of multiple fully-connected layers followed by activation functions, such as ReLU, to introduce non-linearity. Residual, or skip, connections are also included to bypass both the multi-head attention mechanism and the FFN. They are included, along with normalization layers, to stabilize training. Figure 4.12 illustrates this encoder architecture, introduced by Vaswani et al. 2017 and used by several other studies for image classification (Dosovitskiy et al. 2020), object detection (Carion et al. 2020), and semantic segmentation (Strudel et al. 2021) tasks.

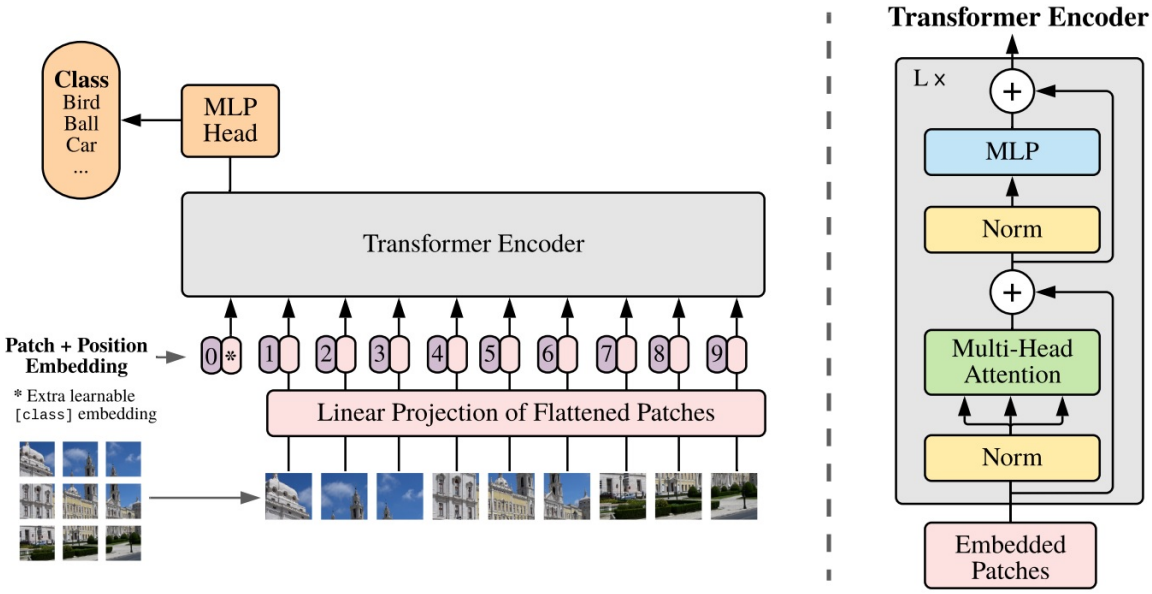


Figure 4.12: Overview of the Transformer Encoder (source: Dosovitskiy et al. 2020)

4.3.5. Detection Transformer (DETR)

The Detection Transformer (DETR), proposed by Carion et al. 2020, is an innovative object detection model that deviates from the traditional approaches to this task. Discarding region proposal networks (Ren et al. 2015) and anchor boxes (Redmon et al. 2016), DETR leverages the power of transformer-based architectures to directly predict bounding boxes and class labels. In this end-to-end solution, the object detection is converted to a sequence-to-sequence task using an encoder-decoder architecture. The input sequence consists of image features and the output is a predefined number of bounding boxes. Figure 4.13 illustrates DETR's architecture.

Instead of linearly projecting image patches to obtain patch embeddings, as described in Section 4.3.3, DETR uses a conventional CNN backbone to extract image features in the shape of a 2D representation vector. Subsequently, this vector is flattened and padded with learnable positional embeddings, as before, to create the encoder's input sequence. Next, a typical transformer encoder (Section 4.3.4) is used to enable the model to capture the global and local context in the input image features. The encoder's output is then passed to DETR's decoder.

The **decoder** is responsible for learning relationships between image pixels and objects, which

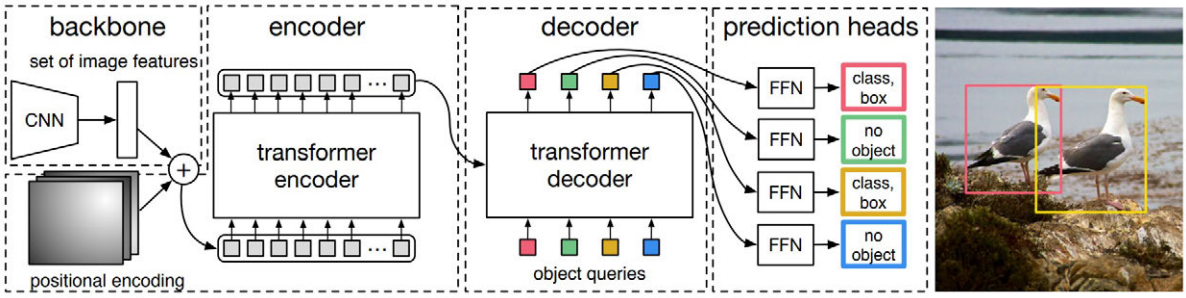


Figure 4.13: Overview of DETR's architecture (source: Carion et al. 2020)

will be used to predict bounding boxes and class labels. Other than the encoder's output, it also takes as input a pre-defined number of learnable positional embeddings, called *object queries*. Similar to the encoder's architecture, the decoder consists of several multi-head self-attention layers and feed-forward networks. Its architecture is illustrated in Figure 4.14, along with the encoder's. The decoder's outputs are subsequently passed to FFNs (*prediction heads*) to generate predictions for object class labels and bounding boxes.

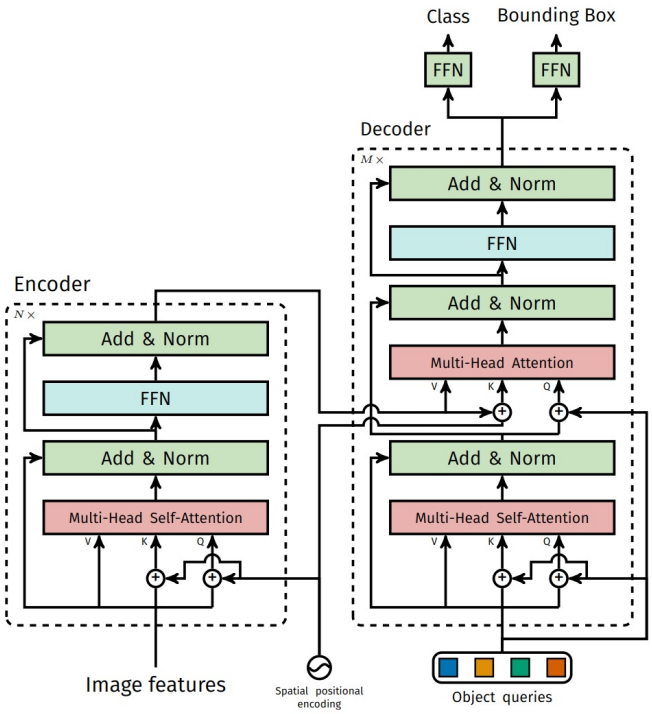


Figure 4.14: Overview of DETR's encoder-decoder architecture (source: Carion et al. 2020)

This direct-set prediction of the objects in images constitutes DETR a suitable solution for **end-to-end chess recognition** (Chapter 2). However, instead of bounding boxes, we want to predict object (piece) positions relative to the chessboard grid in the image. Thus, the prediction heads are modified to output a set of (x, y) coordinates, which are then translated to chessboard positions, along with the object's class.

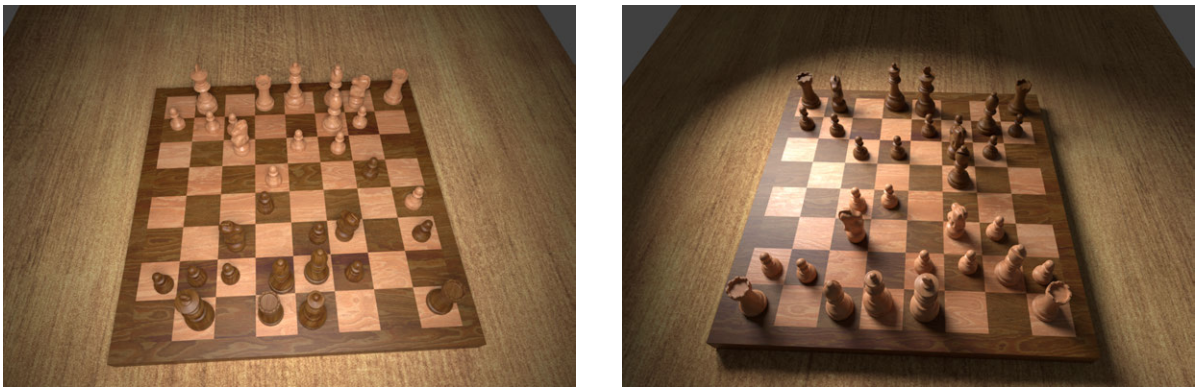
5

Datasets

In this chapter, we will present in more detail the two datasets used in the experiments of Chapter 2. First, we have the Blender dataset used by Georg Wölflein and Arandjelović 2021. Then, we present our new dataset for chess recognition, *ChessReD*.

5.1. Blender dataset

The Blender dataset is a dataset of synthetic images of chessboards rendered in Blender (Community 2018). The piece configurations in the images were randomly selected from a publicly available dataset of chess games played by Magnus Carlsen (Mentor 2020). In particular, 2% of all chess positions in the dataset were sampled, amounting to a total of $4,888$ distinct chess configurations. The dataset provides annotations regarding the pixel coordinates of the chessboard's four corner points in the rendered images, the FEN (Forsyth-Edwards Notations) descriptions of the pieces on the board in each rendered image, the color of the current player, bounding boxes, and detailed information about the camera angle and lighting mode.



(a) Flash **(b)** Spotlight

Figure 5.1: Samples of the Blender dataset (Wölflein and Arandjelovic 2021)

In each rendered image, the pieces were not positioned in the center of the squares, but rather they were randomly rotated and positioned with a random offset to emulate real-world conditions. The camera was aligned to point directly at the center of the chessboard from the perspective of a player's view. The θ angle, between the camera and the board's surface, was selected to be in the range of 45 to 60 degrees to ensure maximum visibility of the pieces. An offset in the x-component of the camera angle was used to introduce some variation. Regarding the lighting condition, a random choice was made between two modes: camera flash and spotlights. Figure 5.1 illustrates two samples of the

Blender dataset with these variations. More information about the dataset can be found in Wölflein and Arandjelovic 2021.

5.2. Chess Recognition Dataset (ChessReD)

ChessReD is a novel chess dataset specifically designed for chess recognition. It is the first comprehensive dataset consisting of real images of chess positions, along with proper annotations for the task. It comprises a diverse collection of images captured using smartphone cameras. We opted to use this type of camera sensor to ensure real-world applicability. A detailed description of the dataset is provided in the corresponding section of Chapter 2. Thus, in this section, we will provide some additional information, statistics, and figures for the dataset.

5.2.1. Physical Chessboard Properties

A plastic, foldable, single-weighted club chess set, that meets competition standards, was used for the images in ChessReD. It contains black and white pieces on brown and white squares. The dimensions of the chessboard are 44cm $\times$ 44cm, with the size of each square being 5.5cm $\times$ 5.5cm. There are borders of 4cm in length on each side with chessboard coordinates in algebraic notation. The king's height is 9.5cm and the weight of the chess set is approximately 0.44kg.

5.2.2. Annotation Statistics

ChessReD contains $10,800$ chessboard images with $10,242$ unique piece configurations. While it is rare, the dataset includes duplicate configurations, mainly due to early game moves and the 100 images of the starting position of each game. However, because of the variability in the viewing angle of the camera, the camera model, and the lighting conditions that differ even between images with the same configuration, we decided to keep these duplicates in the dataset.

The histogram of Figure 5.2 illustrates the frequency distribution of the number of pieces on the chessboard across ChessReD. The x-axis represents the number of pieces on the board, while the y-axis displays the number of images. Given that every chess game starts with 32 pieces on board and rarely ends in a draw with only the two kings remaining, it is reasonable that the frequency for these two configurations to be the highest and the lowest, respectively.

Figure 5.3 illustrates the square occupancy heatmap for ChessReD. The most occupied square in our dataset is $g2$, with an occupancy rate of approximately 65.3%, followed by $g7$ and $f7$, with about 65% each. The least occupied square, which is occupied in $1,413$ out of $10,800$ images, or about 13%, is square $g4$.

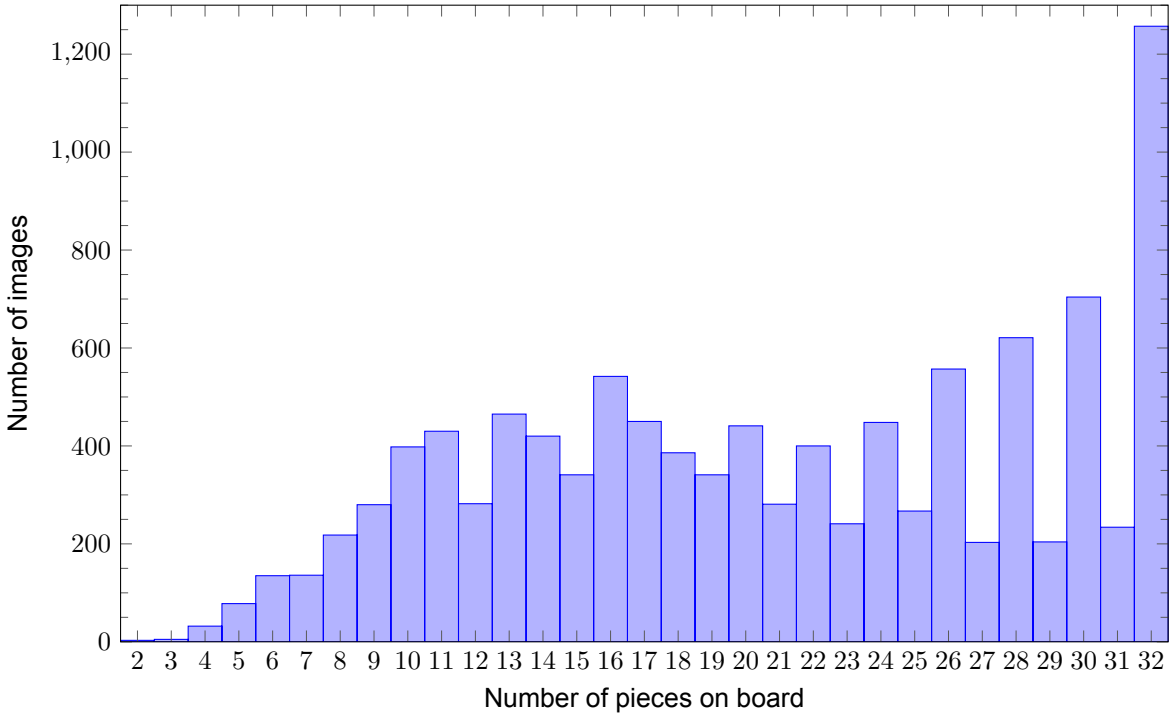


Figure 5.2: Distribution of chessboard configurations in ChessReD

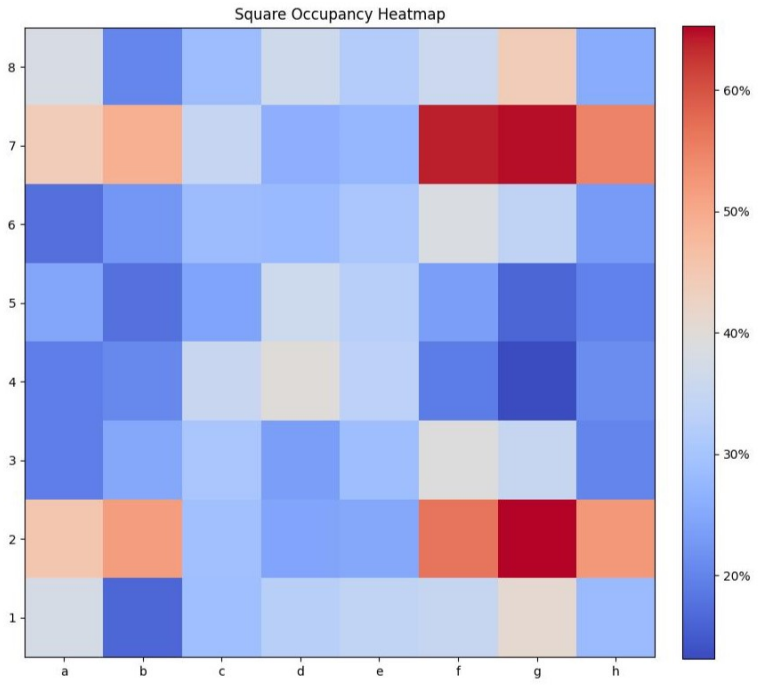


Figure 5.3: Square occupancy heatmap in ChessReD

6

Experimental results

In this chapter, we will discuss some additional results and observations from our experiments, for both the classification (Section 6.1) and the relative object detection (Section 6.2) approaches.

6.1. Classification Approach

As mentioned in Chapter 2, in the classification approach we employ a model to perform multi-label multi-class classification. While ResNeXt (Xie et al. 2017) performed best on ChessReD's validation dataset, a number of models were trained on the same task. Table 6.1 presents the accuracy of the trained models on the validation set.

Model	Boards with no mistakes (%)
ResNet50	5.07%
ConvNext_base	5.66%
EfficientNetV2	9.81%
Wide_ResNet101	12.04%
ResNeXt101	**14.47%**

Table 6.1: Performance of selected classification models on ChessReD's validation dataset

We also evaluated the model's ability to detect the position of the pieces on the board, regardless of their type. For this reason, a new metric, *occupancy accuracy*, that determines the percentage of boards in which the predicted chess configurations contain the same occupied squares as in the target configurations was defined. The ResNeXt model achieved a 42.69% occupancy accuracy, which means that in 909 out of 2129 test images, the model was able to successfully predict the occupied squares, while the correct types of pieces were detected in only 325 of them. Upon further qualitative investigation, the main factor that led to the misclassification of the pieces' type was the occlusions between pieces, followed by top viewing angles that concealed some of the pieces' unique characteristics (*e.g.* the queen's crown).

6.2. Relative Object Detection

For the relative object detection approach of Chapter 2, we employed variations of the DETR model (Carion et al. 2020). We trained the models both from scratch and by using pre-trained weights for different backbone networks (ResNet and ResNeXt variants). Regardless, the training was not successful, with neither of the trained variants being able to perform chess recognition. Although the validation loss was reduced (Figure 6.1), which is a combination of a cross entropy-loss for the classes and an L1 loss for the coordinates, the recognition accuracy (*i.e.* percentage of boards with no mistakes) on the validation set remained zero throughout the training process. Additionally, while 61.45% of the positions

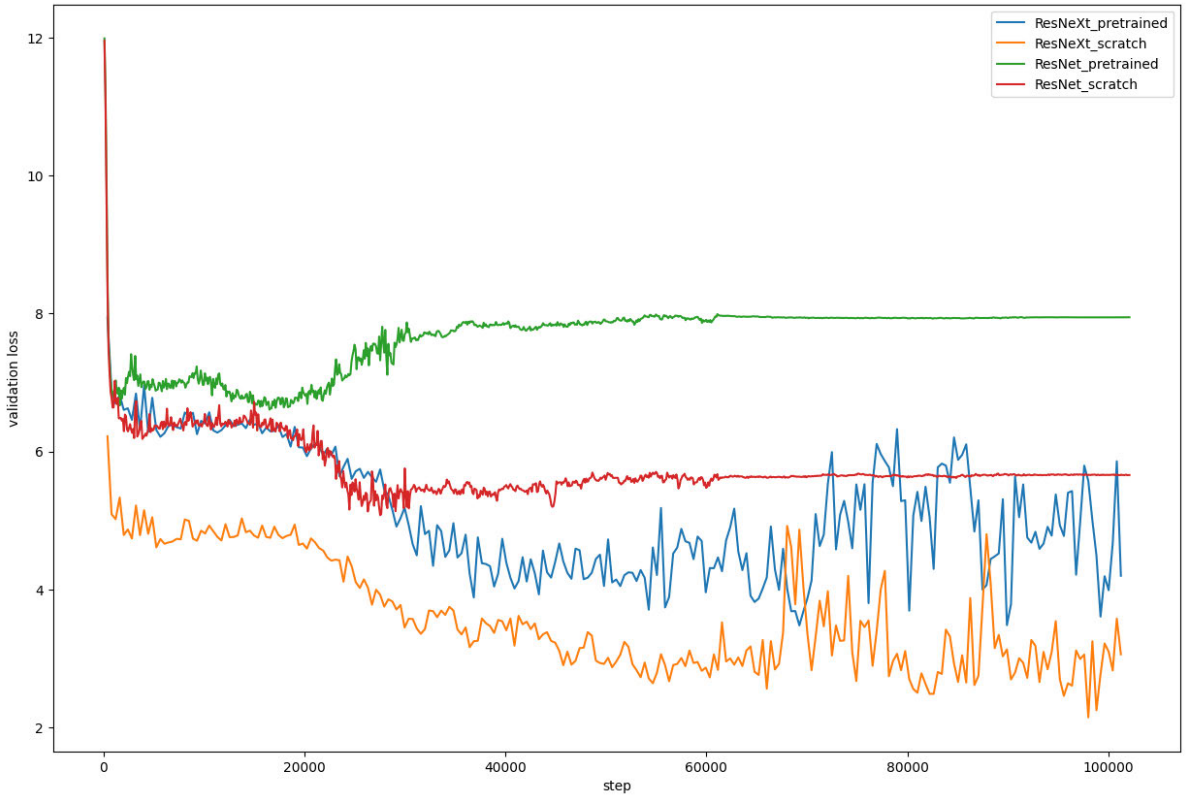


Figure 6.1: DETR variants' performance on ChessReD's validation dataset (the variants are denoted as *[backbone]_[weights]*, with the 101-layer architectures selected for all backbones)

of the chess pieces in the validation dataset were correctly predicted for the best performing DETR variant (ResNeXt101 backbone, trained from scratch), the occupancy accuracy is 0% since there is an average of 11.5 miss-positioned pieces per image.

Bibliography

[1] Augustin Cauchy et al. “Méthode générale pour la résolution des systemes d’équations simultanées”. In: *Comp. Rend. Sci. Paris* 25.1847 (1847), pp. 536–538 (cit. on p. 27).

[2] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133 (cit. on p. 23).

[3] Claude Elwood Shannon. “A mathematical theory of communication”. In: *The Bell system technical journal* 27.3 (1948), pp. 379–423 (cit. on p. 27).

[4] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *The annals of mathematical statistics* (1951), pp. 400–407 (cit. on pp. 23, 27).

[5] Jack Kiefer and Jacob Wolfowitz. “Stochastic estimation of the maximum of a regression function”. In: *The Annals of Mathematical Statistics* (1952), pp. 462–466 (cit. on p. 27).

[6] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 23).

[7] Paul VC Hough. *Method and means for recognizing complex patterns*. US Patent 3,069,654. 1962 (cit. on p. 16).

[8] Godfrey N Lance and William Thomas Williams. “A general theory of classificatory sorting strategies: 1. Hierarchical systems”. In: *The computer journal* 9.4 (1967), pp. 373–380 (cit. on p. 18).

[9] James MacQueen et al. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297 (cit. on p. 18).

[10] Irwin Sobel, Gary Feldman, et al. “A 3x3 isotropic gradient operator for image processing”. In: *a talk at the Stanford Artificial Project in* (1968), pp. 271–272 (cit. on p. 12).

[11] Judith MS Prewitt et al. “Object enhancement and extraction”. In: *Picture processing and Psychopictorics* 10.1 (1970), pp. 15–19 (cit. on p. 12).

[12] Richard O Duda and Peter E Hart. “Use of the Hough transformation to detect lines and curves in pictures”. In: *Communications of the ACM* 15.1 (1972), pp. 11–15 (cit. on p. 16).

[13] Kunihiko Fukushima. “Cognitron: A self-organizing multilayered neural network”. In: *Biological cybernetics* 20.3-4 (1975), pp. 121–136 (cit. on p. 25).

[14] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological cybernetics* 36.4 (1980), pp. 193–202 (cit. on pp. 23, 28).

[15] Martin A Fischler and Robert C Bolles. “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography”. In: *Communications of the ACM* 24.6 (1981), pp. 381–395 (cit. on pp. 17, 21).

[16] John J Hopfield. “Neural networks and physical systems with emergent collective computational abilities.” In: *Proceedings of the national academy of sciences* 79.8 (1982), pp. 2554–2558 (cit. on p. 23).

[17] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137 (cit. on p. 18).

[18] John Canny. “A computational approach to edge detection”. In: *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), pp. 679–698 (cit. on p. 12).

[19] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors”. In: *nature* 323.6088 (1986), pp. 533–536 (cit. on pp. 23, 25, 27).

[20] Yann LeCun et al. “Backpropagation applied to handwritten zip code recognition”. In: *Neural computation* 1.4 (1989), pp. 541–551 (cit. on pp. 23, 28).

[21] Jeffrey L Elman. “Finding structure in time”. In: *Cognitive science* 14.2 (1990), pp. 179–211 (cit. on p. 23).

[22] Nahum Kiryati, Yuval Eldar, and Alfred M Bruckstein. “A probabilistic Hough transform”. In: *Pattern recognition* 24.4 (1991), pp. 303–316 (cit. on p. 16).

[23] Martin Ester et al. “A density-based algorithm for discovering clusters in large spatial databases with noise”. In: *kdd*. Vol. 96. 34. 1996, pp. 226–231 (cit. on pp. 18, 19).
[24] Sepp Hochreiter. “The vanishing gradient problem during learning recurrent neural nets and problem solutions”. In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 6.02 (1998), pp. 107–116 (cit. on p. 25).
[25] Jiri Matas, Charles Galambos, and Josef Kittler. “Robust detection of lines using the progressive probabilistic hough transform”. In: *Computer vision and image understanding* 78.1 (2000), pp. 119–137 (cit. on p. 16).
[26] Erich L Lehmann and George Casella. *Theory of point estimation*. Springer Science & Business Media, 2006 (cit. on p. 26).
[27] Elan Dubrofsky. “Homography estimation”. In: *Diplomová práce. Vancouver: Univerzita Britské Kolumbie* 5 (2009) (cit. on p. 21).
[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012) (cit. on p. 32).
[29] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on p. 28).
[30] Kaiming He et al. “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034 (cit. on p. 25).
[31] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems* 28 (2015) (cit. on p. 34).
[32] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on pp. 28, 31).
[33] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788 (cit. on p. 34).
[34] Sergey Zagoruyko and Nikos Komodakis. “Wide residual networks”. In: *arXiv preprint arXiv:1605.07146* (2016) (cit. on p. 31).
[35] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708 (cit. on p. 31).
[36] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017) (cit. on pp. 23, 32–34).
[37] Saining Xie et al. “Aggregated residual transformations for deep neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500 (cit. on pp. 28, 31, 32, 39).
[38] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Stichting Blender Foundation, Amsterdam, 2018. URL: http://www.blender.org (cit. on p. 36).
[39] Nebojša Bačanin Džakula et al. “Convolutional neural network layers and architectures”. In: *Sinteza 2019-International Scientific Conference on Information Technology and Data Related Research*. Singidunum University. 2019, pp. 445–451 (cit. on p. 28).
[40] Jean-Baptiste Cordonnier, Andreas Loukas, and Martin Jaggi. “On the relationship between self-attention and convolutional layers”. In: *arXiv preprint arXiv:1911.03584* (2019) (cit. on p. 32).
[41] Prajit Ramachandran et al. “Stand-alone self-attention in vision models”. In: *Advances in neural information processing systems* 32 (2019) (cit. on p. 32).
[42] Nicolas Carion et al. “End-to-end object detection with transformers”. In: *European conference on computer vision*. Springer. 2020, pp. 213–229 (cit. on pp. 34, 35, 39).
[43] Alexey Dosovitskiy et al. “An image is worth 16x16 words: Transformers for image recognition at scale”. In: *arXiv preprint arXiv:2010.11929* (2020) (cit. on pp. 32, 34).
[44] PGN Mentor. *Magnus Carlsen Chess Games*. PGN Mentor. Online, 2020. URL: https://www.pgnmentor.com/players/Carlsen/ (cit. on p. 36).
[45] Huo Yingge, Imran Ali, and Kang-Yoon Lee. “Deep neural networks on chip-a survey”. In: *2020 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE. 2020, pp. 589–592 (cit. on p. 30).

[46] Robin Strudel et al. “Segmenter: Transformer for semantic segmentation”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 7262–7272 (cit. on p. 34).
[47] G Wölflein and O Arandjelovic. “Dataset of Rendered Chess Game State Images”. In: *Open Science Framework* (2021) (cit. on pp. 36, 37).
[48] Georg Wölflein and Ognjen Arandjelović. “Determining chess game state from an image”. In: *Journal of Imaging* 7.6 (2021), p. 94 (cit. on p. 36).