

# MSCI DISSERTATION



University of  
St Andrews

## Vision-based inference of chess positions

*Author:*  
Georg WÖLFLEIN                    *Supervisor:*  
Dr. Ognjen ARANDJELOVIĆ

29th December 2020

# **Abstract**

'Tis very abstract indeed.

# Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 16775 words long.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

*Georg Wöllein*

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Context survey . . . . .	2
1.2	Objectives . . . . .	6
1.3	Software engineering process . . . . .	6
1.4	Ethics . . . . .	6
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Supervised learning . . . . .	7
2.2	Artificial neural networks . . . . .	9
2.3	Convolutional neural networks . . . . .	15
2.4	Transfer learning . . . . .	20
<b>3</b>	<b>Data synthesis</b>	<b>22</b>
3.1	Chess positions . . . . .	22
3.2	Three-dimensional renders . . . . .	22
3.3	Automated labelling . . . . .	26
3.4	Splitting the dataset . . . . .	26
<b>4</b>	<b>Recognising chess positions</b>	<b>29</b>
4.1	Board localisation . . . . .	30
4.2	Occupancy classification . . . . .	43
4.3	Piece classification . . . . .	49
4.4	Producing a prediction . . . . .	52
<b>5</b>	<b>Adapting to new chess sets</b>	<b>54</b>
5.1	Dataset . . . . .	55
5.2	Training . . . . .	56
<b>6</b>	<b>Implementation</b>	<b>61</b>
6.1	Configuration management (recap) . . . . .	64
6.2	Chess recognition system ( <i>chesscog</i> ) . . . . .	65
6.3	Web app ( <i>chesscog-app</i> ) . . . . .	65
<b>7</b>	<b>Evaluation</b>	<b>68</b>
7.1	Achievement of objectives . . . . .	69
7.2	Critical appraisal . . . . .	69
<b>8</b>	<b>Conclusion</b>	<b>70</b>
8.1	Future work . . . . .	70

*CONTENTS*

---

<b>Acronyms</b>	<b>71</b>
<b>Bibliography</b>	<b>72</b>
<b>A Testing summary</b>	<b>77</b>
<b>B User manual: recap package</b>	<b>78</b>
B.1 Documentation . . . . .	78
<b>C User manual: chesscog</b>	<b>79</b>
C.1 Documentation . . . . .	79
C.2 Data synthesis . . . . .	79
<b>D User manual: web app</b>	<b>80</b>
<b>E Ethics self-assessment form</b>	<b>81</b>

# List of Figures

2.1	An artificial neural network with six neurons that could be used to decide a classification problem. . . . .	9
2.2	Plots of the two most common activation functions. . . . .	10
2.3	A SLP with two inputs and three outputs. . . . .	11
2.4	A MLP with three inputs, two hidden layers, and two outputs. .	13
2.5	An illustration gradient descent training on an error-weight surface.	14
2.6	Illustration of a convolution operation with a $2 \times 2$ kernel. . . . .	17
2.7	Illustration of a convolution operation acting on two input channels.	18
2.8	Illustration of maximum pooling for $k = s = 2$ . . . . .	20
2.9	Illustration of the transfer learning process for domain adaption in deep neural networks. . . . .	21
3.1	The chessboard coordinate system. . . . .	23
3.2	Side view of the camera setup for the scenario where it is white to move. . . . .	24
3.3	Two samples from the synthesised dataset showing both types of lighting. . . . .	25
3.4	Overhead view of the chessboard with two spotlights. . . . .	25
3.5	Visualisation of the automatically generated labels. . . . .	27
3.6	Visual representation of the dataset split. . . . .	28
4.1	Overview of the chess recognition pipeline. . . . .	29
4.2	The process of determining the intersection points on the chessboard. . . . .	31
4.3	A line in Hesse normal form. . . . .	31
4.4	An example of how lines in image space are represented in $(\rho, \theta)$ space. . . . .	32
4.5	Projection of four intersection points from the original to the warped image. . . . .	34
4.6	A sample from the training set with an incorrectly identified line.	35
4.7	The original image is warped using the computed homography matrix $\mathbf{H}$ . . . . .	38
4.8	Horizontal gradient intensities calculated on the warped image in order to detect vertical lines. . . . .	41
4.9	The identified corner points are projected back from the warped image onto the original image using the inverse homography matrix.	42
4.10	An example illustrating why an immediate piece classification approach is inclined to reporting false positives. . . . .	43

*LIST OF FIGURES*

---

4.11	The process of obtaining samples for occupancy classification from a chessboard image. . . . .	44
4.12	Architecture of the CNN (100, 3, 3, 3) network for occupancy classification. . . . .	45
4.13	Loss and accuracy during training on both the training and validation sets for the CNN (100, 3, 3, 3) model. . . . .	46
4.14	Loss and accuracy during training on both the training and validation sets for the ResNet model. . . . .	47
4.15	The four samples that the ResNet model misclassified in the validation set. . . . .	47
4.16	The normals of the chessboard surface converge to a single vanishing point which is below the image. . . . .	50
4.17	A random selection of six samples of white queens in the training set. . . . .	51
4.18	Loss and accuracy during training on both the training and validation sets for the InceptionV3 model. . . . .	52
5.1	The starting position on the board. . . . .	55
5.2	The training dataset used for the transfer learning approach, consisting of only two samples. . . . .	55
5.3	Illustration of the shear transform with $\lambda = 1/2$ . . . . .	57
5.4	The augmentation pipeline applied to an input image. . . . .	58
5.5	Loss and accuracy in fine-tuning the occupancy and piece classifiers on the new dataset. . . . .	59
6.1	Overview of CI/CD pipelines. . . . .	63
6.2	Schematic overview of the infrastructure pertaining to the web app. . . . .	66
7.1	Inference time benchmarks of the chess recognition pipeline on the test set. . . . .	69

# List of Tables

4.1	Performance of all occupancy classification models on the validation set. . . . .	48
4.2	Performance of all piece classifiers on the validation set. . . . .	51
4.3	Performance of the chess recognition pipeline on the training and validation sets. . . . .	53
5.1	Performance of the chess recognition pipeline from chapter 4 on the transfer learning dataset without fine-tuning. . . . .	56
5.2	Performance of the fine-tuned chess recognition pipeline on the transfer learning dataset. . . . .	60
7.1	Performance of the chess recognition system on the test dataset. . . . .	68

# Notation

This report follows typical notation conventions established in the deep learning community, particularly those in line with Goodfellow *et al.* [1].

$a$	a scalar (integer or real)
$\mathbf{a}$	a vector
$\mathbf{A}$	a matrix
$\mathcal{A}$	a set
$a$	a scalar random variable
$\mathbf{a}$	a vector random variable
$\mathbf{A}^\top$	transpose of matrix $\mathbf{A}$
$\mathbf{I}_n$	$n \times n$ identity matrix
$\mathbf{I}$	identity matrix (dimensionality implied by context)
$\mathbf{A} \odot \mathbf{B}$	pointwise (Hadamard) product
$\mathbf{A} * \mathbf{B}$	convolution <sup>1</sup>
$\mathcal{N}(\mu, \sigma^2)$	normal distribution
$a \sim P$	random variable $a$ follows distribution $P$
$f(\cdot)$	scalar-valued function
$\mathbf{f}(\cdot)$	vector-valued function
$f \circ g$	composition of the functions $f$ and $g$
$ \cdot $	absolute value
$ \mathcal{A} $	cardinality of set $\mathcal{A}$
$\ \cdot\ _1$	$\ell_1$ norm (Manhattan distance)
$\ \cdot\ _2$	$\ell_2$ norm (Euclidean distance)
$\mathbb{R}$	the set of real numbers
$\mathbb{Z}$	the set of integers
$\frac{\partial y}{\partial x}$	derivative of $y$ with respect to $x$
$\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$	Jacobian vector $\mathbf{J} \in \mathbb{R}^n$ of $f : \mathbb{R}^n \rightarrow \mathbb{R}$
$b_j^{(l)}$	the $j^{\text{th}}$ unit's bias in layer $l$
$w_{i,j}^{(l)}$	the weight of the connection from unit $i$ in layer $l - 1$ to unit $j$ in layer $l$
$\mathbf{W}^{(l)}$	the weight matrix of layer $l$
$\mathbf{b}^{(l)}$	the bias vector of layer $l$

---

<sup>1</sup>Strictly speaking, we use  $*$  to denote cross-correlation, as explained in section 2.3.1.

*Improving your weaknesses has the potential for the greatest gains.*

Garry Kasparov

# 1

## Introduction

The epigraph above – remarked by former World Chess Champion Garry Kasparov in his most recent book [2] – is a profound observation that applies even outside of chess. With regard to chess in particular, Kasparov implies that you must identify your mistakes and weaknesses in order to improve as a player, and to do so, you must analyse your own games.

Amateur chess players can analyse games they played online without much effort because the moves are recorded automatically. However, to analyse over-the-board games<sup>1</sup>, players must tediously enter the position in the computer piece by piece. A casual over-the-board game between two friends will often reach an interesting position<sup>2</sup>. After the game, the players will want to analyse that position on a computer, so they take a photo of the position. On the computer, they need to drag and drop pieces onto a virtual chessboard until the position matches the one they had on the photograph, and then they must double-check that they did not miss any pieces.

The main goal of this project is to develop a system that is able to map a photo of a chess position to a structured format that can be understood by chess engines, such as the widely-used Forsyth–Edwards Notation (FEN) [3], in order to automate this laborious task.

---

<sup>1</sup>Usually, players will invest more effort in over-the-board games, both in terms of time and deep thinking. These games will also involve a greater psychological aspect as a result of being able to observe the opponent's expressions. As such, analysing these games should be even more interesting and fruitful.

<sup>2</sup>For example, one of the players might have a few moves that look promising, but is also considering a line with a piece sacrifice. If he decides to play it safe, he will likely want to analyse the piece sacrifice on the computer after the game.

## 1.1 Context survey

This context survey will examine not only past approaches to chess recognition, but also current implementation tools that will be useful for designing such a system.

### 1.1.1 Chess recognition

Determining the game state of a chess board, also known as *chess recognition*, is a problem in computer vision whereby an algorithm is tasked with recovering the configuration of pieces from an image of a chessboard. Early work on chess recognition in the 1990s focused on extracting typeset games from printed material [4]. In recent years, the problem of parsing two-dimensional chess images has effectively been solved using conventional machine learning techniques [5] and deep learning [6], [7]. However, recognising chess positions from physical chessboards as opposed to artificial two-dimensional images poses a much more interesting and challenging problem that finds practical application in chess-playing robots, augmented reality, and aiding amateur chess players<sup>3</sup>.

**Chess robots** Initial research into chess recognition emerged from the development of chess robots that included a camera to detect the human opponent’s moves from a top-down overhead perspective. The difficulty of distinguishing between chess pieces from a bird’s-eye-view due to their similarity is noted in many papers; as a result, chess robots typically implement a three-way classification system that for every square attempts to determine whether it contains a piece, and, if so, that piece’s colour. Various approaches have been explored including employing manual thresholding [9]–[12] and clustering [13] in different colour spaces, as well as differential imaging (classifying based on the per-pixel difference between two images) [14], [15]. Although the *Gambit* robot proposed by Matuszek *et al.* [16] does not require a bird’s-eye view over the chessboard and uses a depth camera to more reliably detect the occupancy of each square, it employs the three-way classification strategy using a linear support vector machine (SVM) to determine the piece colour.

**Chess move recording** Several techniques for recording chess moves from video footage have been proposed that follow a similar three-way occupancy and colour classification scheme, both from a top-down perspective [8], [17] as well as from a camera positioned at an acute angle to the board [18]. However, in any such three-way classification approach, the robot or move recorder requires knowledge of the previous board state in addition to its predictions for each square’s occupancy and piece colour to deduce the last move. While this information is readily available to a chess robot or move recording software, it is not for a chess recognition system that should deduce the position from a single still image. Furthermore, these approaches experience severe shortcomings in terms of their inability to recover once a single move was predicted incorrectly and fail to identify promoted pieces<sup>4</sup> [9].

---

<sup>3</sup>Electronic chess sets are impractical and very costly [8], thus solutions for chess recognition using just a photo of an unmodified chess board are more compelling for amateur chess players.

<sup>4</sup>Piece promotion occurs when a pawn reaches the last rank, in which case the player must choose to promote to a queen, rook, bishop or knight. Evidently, a vision system that can

**Single-image chess recognition** A number of techniques have been developed to address the issue of chess recognition from a single image. Unlike move recording software or chess robots, the squares’ occupancy and colour alone provide insufficient information to determine the position. These techniques must instead implement a classification algorithm for each piece type (pawn, knight, bishop, rook, queen, and king) of each colour which poses a significantly more difficult problem, attracting research mainly in the last five years. From a bird’s-eye view, the pieces are nearly indistinguishable, hence the photo is usually taken at an acute angle to the board. Ding [19] proposes a piece classifier that uses one-versus-rest SVMs trained on scale-invariant feature transform (SIFT) and histogram of oriented gradients (HOG) feature descriptors, achieving an accuracy of 85%. Conversely, Danner and Kafafy [20] as well as Xie *et al.* [21] claim that SIFT and HOG provide inadequate features for the problem of piece classification due to the similarity in texture between chess pieces, and instead focus on the pieces’ outlines. As such, Danner and Kafafy [20] use Fourier descriptors calculated based on the pieces’ contours, but this requires a manually-created database of piece silhouettes. Furthermore, they modify the board colours to red and green instead of black and white, in order distinguish the pieces from the board more easily<sup>5</sup>. On the other hand, Xie *et al.* [21] perform contour-based template matching with an interesting caveat: the camera angle is calculated based on the perspective transformation of the chessboard, and then depending on the angle, different templates are utilised for matching the chess pieces. As part of the same work, Xie *et al.* develop another approach that instead utilises convolutional neural networks (CNNs), but find that their original template-matching technique achieves superior results in terms of speed and accuracy in low-resolution images. However, the reader should note that their CNNs are trained on only 40 images per class, but deep learning methods tend to excel when trained on larger datasets [22].

**Chessboard detection** A prerequisite to any chess recognition system is the ability to detect the location of the chessboard and each of the 64 squares. Once the four corner points have been established, finding the squares is trivial for pictures captured in bird’s-eye view, and only a matter of a simple perspective transformation in the case of other camera positions. While finding the corner points of a chessboard is frequently used for automatic camera calibration due to the regular nature of the chessboard pattern [23], [24], techniques designed for this purpose tend to perform poorly when there are pieces on the chessboard that occlude lines or corners. Some of the aforementioned chess robots [13], [14], [17] as well as the single-image recognition system proposed by Danner and Kafafy [20] circumvent this problem entirely by prompting the user to interactively select the four corner points, but ideally a chess recognition system should be able to parse the position on the board without human intervention. Most approaches for automatic chess grid detection utilise either the Harris corner detector [11], [18] or a form of line detector based on the Hough transform [12], [15], [20], [25]–[28], although other techniques such as template matching [16] and flood fill [8] have been explored. In general, corner-based al-

---

only detect the piece’s colour is unable to detect what it was promoted to.

<sup>5</sup>Similar board modifications have also been proposed as part of chess robots [11] and chess move trackers [8], but any such modification imposes an unreasonable constraint on normal chess games.

gorithms are unable to accurately detect grid corners when they are occluded by pieces, thus line-based detection algorithms appear to be the favoured solution. Such algorithms often take advantage of the geometric nature of the chessboard which allows to compute a perspective transformation of the grid lines that best matches the detected lines [18], [21], [25]. However, lines found in the background of the photo can often cause failure modes. A recent chess grid detection algorithm that is highly successful even on populated boards is described by Xie *et al.* in [28]. They apply several clustering algorithms on the lines detected via a Hough transform in order to find the horizontal and vertical grid lines belonging to the chessboard, and use this algorithm as a preprocessing step in their template-matching piece classification technique [21] described above.

**Chess recognition using CNNs** Ever since Xie *et al.* pioneered the use of CNNs in the domain of chess recognition from monocular images in 2018<sup>6</sup>, a few more techniques have been developed that employ CNNs at various stages in the recognition pipeline. Czyzewski *et al.* [30] achieve an accuracy of 95% on chessboard detection from non-vertical camera angles by designing an iterative algorithm that generates heatmaps over the input image representing the likelihood of each pixel being part of the chessboard. They then employ a CNN to refine the corner points that were found using the heatmap, outperforming the results obtained by Gonçalves *et al.* [13]. Furthermore, they compare a CNN-based piece classification algorithm to the SVM-based solution proposed by Ding [19] and find no notable amelioration, but manage to obtain major improvements by implementing a probabilistic reasoning system that uses the open source Stockfish chess engine [31] as well as chess statistics [32]. Although reasoning techniques were already employed for refining the predictions of chess recognition systems before [20], [26], Czyzewski *et al.* demonstrate the potential of combining information obtained from a chess engine with large-scale chess statistics. Very recently, Mehta and Mehta [33] implemented an augmented reality app using the popular *AlexNet* CNN architecture introduced by Krizhevsky *et al.* [34], achieving promising results. Despite using an overhead camera perspective and not performing any techniques to ensure probable and legal chess positions, Mehta and Mehta achieve an end-to-end accuracy of 93% for the entire chessboard detection and piece classification pipeline, which – to the best of our knowledge – constitutes the current state of the art. However, it is unclear whether this accuracy is reported on the training set or a possible held-out test set.

**Datasets** The lack of adequate datasets for chess recognition has been recognised by many [19], [30], [33]. Although Czyzewski *et al.* [30] published a dataset of chessboard lattice points that are difficult to predict [35], large datasets – especially at the scale required for deep learning – are not available as of now. Using synthesised data in the training set is an efficient means of creating sizable datasets while minimising the manual annotation efforts [29], [30], [36].

<sup>6</sup>Wei *et al.* [29] developed a chess recognition system using a volumetric CNN one year previously, but this approach requires three-dimensional chessboard data obtained from a depth camera. Their approach achieved a per-class accuracy over 90% except for the “king” class, was trained on computer-aided design (CAD) models, and evaluated on real three-dimensional images (point clouds) of a chessboard.

Czyzewski *et al.* distort some input images in order to simulate different camera perspectives on the chessboard corners. However, a more promising method seems to be the use of three-dimensional models. Wei *et al.* [29] synthesise point cloud data for their volumetric CNN directly from three-dimensional chess models and Hou [36] use renderings of three-dimensional models as input. Yet Wei *et al.* [29]’s approach works only if the chessboard was captured with a depth camera and Hou [36] presents a chessboard recognition system using a simple artificial neural network (ANN) that is not convolutional and hence achieves an accuracy of only 72%.

### 1.1.2 Implementation tools

In both academia and industry, Python is the de facto standard programming language for machine learning. A 2019 analysis conducted on the world-leading software development platform GitHub found that Python is the most popular language for open source machine learning repositories [37]. Python is a simple yet versatile language that natively supports different programming paradigms (imperative, functional, object-oriented, and more). It is often called an interpreted language<sup>7</sup> because it is dynamically typed and performs automatic memory management (garbage collection) which generally facilitates shorter code than compiled languages such as C or Java, but also means that pure-Python implementations of data-intensive algorithms will usually not be as efficient. One of the most fundamental packages, NumPy, implements very efficient array manipulation operations that, although specified in Python, are carried out at a lower level for performance.

NumPy is just one piece of Python’s rich ecosystem of packages that are maintained by open-source contributors in the scientific and engineering community. The two main frameworks for machine learning are TensorFlow by Google and PyTorch by Facebook. At their core, both frameworks facilitate the computation of mathematical operations on tensors<sup>8</sup>, offering support for hardware acceleration via graphics processing units (GPUs) and providing parallelisation strategies for distributed computing which is especially potent in the context of machine learning where many operations fit the single instruction, multiple data (SIMD) pattern. A TensorFlow program is specified as a directed *computational graph* where nodes represent operations and edges represent their inputs and outputs (data tensors) [38]. In the new TensorFlow 2, this graph does not need to be explicitly constructed in the code anymore but is created on the fly which is known as *eager execution*, thereby providing the user with a simpler interface similar to NumPy. The slightly younger PyTorch framework [39] provided dynamic computation graphs and a NumPy-like interface from the outset at its initial release in 2016, and more recently added support for static computational graphs. Hence the newest versions of both frameworks provide similar programming interfaces and computational capabilities. They also facilitate the automatic computation of gradients which is useful for training neural networks. In the research community, PyTorch recently overtook TensorFlow in terms of popularity; at every top machine learning conference since 2019, the

---

<sup>7</sup>There is nuance associated with this statement, but Python certainly exhibits more traits of an interpreted than a compiled language.

<sup>8</sup>Tensors are essentially a generalisation of scalars, vectors, and matrices. They can be thought of as representing a multi-dimensional array.

majority of submissions with code were implemented in PyTorch [40].

## 1.2 Objectives

### 1.2.1 Primary

1. Perform a literature review of available methods for parsing chess positions from photos.
2. Develop an algorithm for detecting the corners of the chessboard as well as the squares.
3. Develop an algorithm for recognising the chess pieces.
4. Develop an algorithm that uses the outputs from (2) and (3) in order to compute a probability distribution over each piece in each square.
5. Evaluate the performance of the developed algorithms.

explain  
extent of  
achieving  
them

### 1.2.2 Secondary

1. Create a large labelled dataset of synthesised chessboard images using 3D models.
2. Implement an algorithm that takes as input the raw probability distribution of each piece in each square and outputs a likely Forsyth–Edwards Notation [3] (FEN) description.
3. Implement a simple web API that performs the inference pipeline for an input image, returning the FEN description.

### 1.2.3 Tertiary

1. Develop a web app that allows the user to upload an image of the chess board to obtain the FEN description.
2. Employ transfer learning to demonstrate how the system can adapt to new chess sets.

Note: final tertiary objective was not an objective in the outset; was added during the project, as it became clear that employing a transfer learning approach would make the system significantly more useful.

## 1.3 Software engineering process

todo

## 1.4 Ethics

There are no ethical issues raised by this project, as indicated in the signed ethics form in appendix E.

*To become good at anything, you have to know how to apply basic principles. To become great at it, you have to know when to violate those principles.*

Garry Kasparov

# 2

## Background

This chapter will introduce some of the basic concepts involved in ANNs, CNNs, and transfer learning as they relate to this project. Readers familiar with this subject matter may skip this chapter and resume at chapter 3.

### 2.1 Supervised learning

At its core, the purpose of an ANN is to infer a function that maps some input to some output, based on sample input-output pairs. In machine learning, we call this a *supervised learning* task, and there are a great number of machine learning models (not only ANNs) that have been developed for this task. We shall briefly examine the two main disciplines within supervised learning: *regression* and *classification*.

#### 2.1.1 Regression

A regression model captures the relationship between multiple input variables and one output variable. As such, it can be defined a mathematical function of the form  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  given by

$$f(\mathbf{x}) = \hat{y} = y + \epsilon \quad (2.1)$$

that models the relationship between an  $n$ -dimensional feature vector  $\mathbf{x} \in \mathbb{R}^n$  of independent (*input*) variables and the dependent (*output*) variable  $y \in \mathbb{R}$ . Given a particular  $\mathbf{x}$ , the model will produce a *prediction* for  $y$  which we shall denote  $\hat{y}$ . Here, the additive error term  $\epsilon$  represents the discrepancy between  $y$  and  $\hat{y}$ , i.e. the difference between the predicted and observed output.

A labelled dataset for a regression task consists of  $m$  tuples of the form  $\langle \mathbf{x}_i, y_i \rangle$  for  $i = 1, \dots, m$ . For each feature vector  $\mathbf{x}_i$  (a row vector), the corres-

ponding  $y_i$  represents the observed output, or *label* [41]. We use the vector

$$\mathbf{y} = [y_1 \ y_2 \ \cdots \ y_m]^\top \quad (2.2)$$

to denote all the labelled outputs in the dataset, and the  $m \times n$  matrix

$$\mathbf{X} = [\mathbf{x}_1 \ \mathbf{x}_2 \ \cdots \ \mathbf{x}_m]^\top \quad (2.3)$$

for representing the corresponding feature vectors.

### 2.1.2 Classification

Classification is a task that finds greater applicability within this project. As the name implies, a classification model tries to determine to which category each input sample belongs from a predefined set of classes  $\mathcal{C}$ . To ease notation, we shall let  $\mathcal{C} = \{1, 2, \dots, C\}$  where  $C$  is the total number of classes. In practical terms, the elements in  $\mathcal{C}$  could represent any type of mathematical or non-mathematical object, and in that case we only require a one-to-one mapping (bijection) from those objects to  $\mathcal{C}$ . Furthermore, in the context of this report, each input sample can only belong to one class, and we will interpret the output of the model to represent the probabilities of the input sample belonging to each of the classes  $\mathcal{C}$ . Therefore, a classification model is represented by a vector-valued function  $\mathbf{f} : \mathbb{R}^n \rightarrow [0, 1]^{|\mathcal{C}|}$  instead of a scalar function as in eq. (2.1), namely

$$\mathbf{f}(\mathbf{x}) = \hat{\mathbf{y}} = \mathbf{y} + \boldsymbol{\epsilon}. \quad (2.4)$$

Here,  $\mathbf{f}(\mathbf{x})$  actually represents a probability mass function (PMF) where the  $i^{\text{th}}$  component  $\hat{y}_i$  of the output vector  $\hat{\mathbf{y}}$  represents the probability that the input sample  $\mathbf{x}$  belongs to the class  $i \in \mathcal{C}$ . It follows that

$$\sum_{i=1}^{|\mathcal{C}|} \hat{y}_i = 1. \quad (2.5)$$

The observed output  $\mathbf{y}$  is a row vector that uses a one-hot encoding (OHE) to represent the sample's class. This means that for a given class  $c \in \mathcal{C}$ , the components of  $\mathbf{y}$  are given by

$$y_i = \begin{cases} 1 & i = c \\ 0 & \text{otherwise} \end{cases},$$

which retains the property described in eq. (2.5). Since  $\mathbf{f}$  represents a PMF, the one-hot encoded vector essentially states that the probability of class  $c$  is 100% and all other classes have a probability of 0%. Notice that our outputs are now the vectors  $\mathbf{y}_i$  instead of the scalars  $y_i$  in the regression task. Hence a labelled classification dataset will consist of  $m$  tuples of the form  $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$ , meaning that instead of the vector  $\mathbf{y}$  from section 2.1.2, we must use a matrix

$$\mathbf{Y} = [\mathbf{y}_1 \ \mathbf{y}_2 \ \cdots \ \mathbf{y}_m]^\top \quad (2.6)$$

to denote the targets. Each row in  $\mathbf{Y}$  represents the one-hot encoded class label for that sample. The input matrix  $\mathbf{X}$  remains as defined in eq. (2.3).

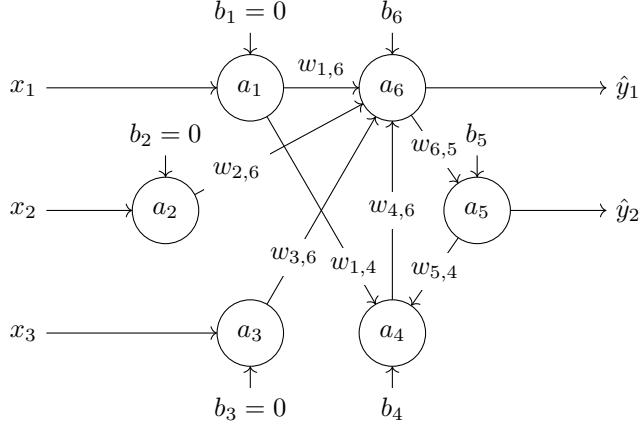


Figure 2.1: An artificial neural network with six neurons that could be used to decide a classification problem. The inputs  $x_1, x_2, x_3$  are set as the activations of the first three units (thus  $b_1, b_2, b_3$  have no effect) and the outputs  $\hat{y}_1, \hat{y}_2$  are the activations received from the final two units. Only some weights are shown in the diagram.

## 2.2 Artificial neural networks

ANNs take inspiration from the human brain and can be regarded as a set of interconnected neurons. More formally, an ANN is a directed graph of  $n$  neurons (referred to as *nodes* or *units*) with weighted edges (*links*). Each link connecting two units  $i$  and  $j$  is directed and associated with a real-valued weight  $w_{i,j}$ .

A particular unit  $i$ 's *excitation*, denoted  $z_i$ , is calculated as the weighted sum

$$z_i = \sum_{j=1}^n w_{j,i} a_j + b_i \quad (2.7)$$

where  $a_j \in \mathbb{R}$  is another unit  $j$ 's *activation* and  $b_i \in \mathbb{R}$  is the  $i^{\text{th}}$  unit's *bias*. In this model, if there exists no link between unit  $i$  and a particular  $j$  then simply  $w_{i,j} = 0$  and therefore  $j$  will not contribute to  $i$ 's excitation. Figure 2.1 gives an example how such a network could look like.

The unit  $i$ 's activation is its excitation applied to a non-linear *activation function*,  $g : \mathbb{R} \rightarrow \mathbb{R}$ . We have

$$a_i = g(z_i) = g\left(\sum_{j=1}^n w_{j,i} a_j + b_i\right). \quad (2.8)$$

**Activation functions** In its original form in 1943, McCulloch and Pitts defined the neuron as having only binary activation [42]. This means that in our model from eq. (2.8), we would require  $a_i \in \{0, 1\}$  and hence an activation function of the form  $g_{\text{step}} : \mathbb{R} \rightarrow \{0, 1\}$  like the Heaviside step function<sup>1</sup>

$$g_{\text{step}}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}.$$

<sup>1</sup>In fact, McCulloch and Pitts defined the activation to be zero when  $x < \theta$  for a threshold parameter  $\theta \in \mathbb{R}$  and one otherwise, but in our model the bias term  $b_i$  acts as the threshold.

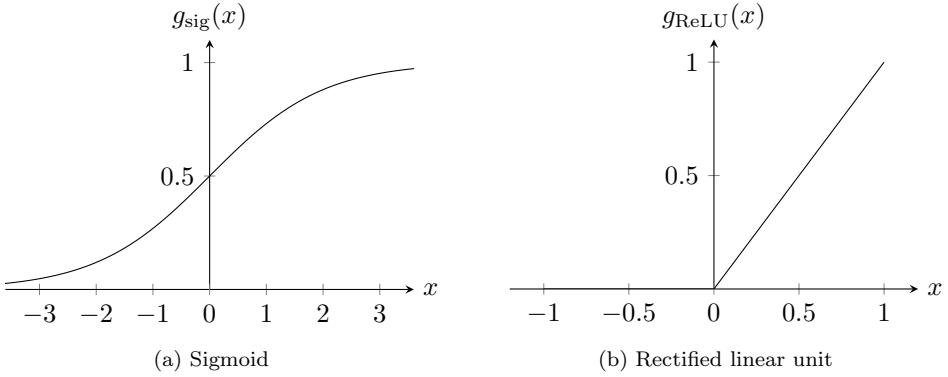


Figure 2.2: Plots of the two most common activation functions.

Commonly used activation functions in modern neural networks include the sigmoid

$$g_{\text{sig}}(x) = \frac{1}{1 + e^{-x}}$$

and the rectified linear unit (ReLU) [43]

$$g_{\text{ReLU}} = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \quad (2.9)$$

which are depicted in fig. 2.2. Unlike  $g_{\text{step}}$ , the range of these activation functions is the real numbers, and the functions themselves are differentiable which is an advantage for being able to use gradient descent to optimise the weights [44, p. 729].

Rectified units do not suffer from the so-called *vanishing gradient effect* [43]. This phenomenon occurs with sigmoid activation functions when they reach high saturation, i.e. when the input is significantly far from zero such that the gradient is almost horizontal (see fig. 2.2(a)), and is especially prevalent in deep neural networks<sup>2</sup>. As a result, the ReLU activation function (or variants thereof) is the most popular choice nowadays. Furthermore, the computational cost of the function itself as well as its gradient is cheap.

### 2.2.1 Feedforward neural networks

Our definition of ANNs so far still is very general and makes virtually no restrictions on the graph of the network. It turns out that it is difficult to propagate activations between neurons when they exhibit cycles, as is the case with the last three units in fig. 2.1. Thus we impose a constraint that the nodes of the network are not allowed to form cycles, and as a result the network becomes a directed acyclic graph (DAG). This class of ANNs is referred to as *feedforward neural networks*.

---

<sup>2</sup>Deep neural networks are ANNs with many layers.

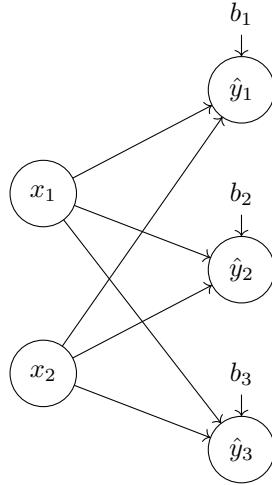


Figure 2.3: A SLP with two inputs and three outputs.

### 2.2.1.1 Single-layer perceptron

The most basic type of feedforward neural network is the single-layer perceptron (SLP). It consists of  $n_0$  input units that are directly connected to  $n_1$  output units, as illustrated in fig. 2.3. There are no connections between input units, and no connections between output units. Likewise, there are no connections from output units to input units. The only connections in the network originate from input units and feed to output units. In fact, that is where the term *feedforward* arises: the network exhibits neither backwards nor intra-layer connections. The connections themselves are of course weighted as in any ANN. Due to the unidirectional nature of the links, we can continue to use the notation  $w_{i,j}$  to denote weights, but now  $i$  refers to the input unit and  $j$  refers to the output unit.

From eq. (2.8), we can compute the values of the three output units in fig. 2.3 as

$$\hat{y}_j = g \left( \sum_{i=1}^n w_{i,j} x_i + b_j \right) \quad (2.10)$$

for  $j = 1, 2, 3$ . Mathematically, a SLP is represented by a function  $\mathbf{f} : \mathbb{R}^{n_0} \rightarrow \mathbb{R}^{n_1}$  that maps an input vector  $\mathbf{x} \in \mathbb{R}^{n_0}$  to an output vector  $\hat{\mathbf{y}} \in \mathbb{R}^{n_1}$ . Let us use the  $n_0 \times n_1$  matrix  $\mathbf{W}$  to contain all weights such that

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n_1} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n_1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_0,1} & w_{n_0,2} & \cdots & w_{n_0,n_1} \end{bmatrix}.$$

and the vector

$$\mathbf{b} = [b_1, b_2, \dots, b_{n_1}]$$

to represent the biases. Since the output vector  $\hat{\mathbf{y}}$  is simply the concatenation of the output units, the summation in eq. (2.10) can be expressed using the dot

product. We finally define the function of a SLP as

$$f_{\mathbf{w}, \mathbf{b}}(\mathbf{x}) = \hat{\mathbf{y}} = \mathbf{g}(\mathbf{w}^\top \mathbf{x} + \mathbf{b}) \quad (2.11)$$

where  $\mathbf{g}(\cdot)$  applies the activation function  $g$  pointwise. The process of computing the activations of the output units given the input units is often called *forward propagation* or *forward pass* [41].

### 2.2.1.2 Multi-layer perceptron

As the name implies, a multi-layer perceptron (MLP) simply stacks multiple SLPs on top of each other, such that each layer's outputs are connected to the next layer's inputs (except for the final layer, where the outputs represent  $\hat{\mathbf{y}}$ ). A MLP with  $L$  layers can be expressed mathematically by composing  $L$  SLPs  $f_1, f_2, \dots, f_L$ :

$$f(\mathbf{x}) = (f_1 \circ f_2 \circ \dots \circ f_L)(\mathbf{x}). \quad (2.12)$$

Each SLP inside the MLP constitutes a logical module which we call a *layer*. Due to the fact that each layer has its own weights  $\mathbf{W}$  and biases  $\mathbf{b}$ , we will use the parenthesised superscript notation  $\mathbf{W}^{(l)}$  and  $\mathbf{b}^{(l)}$  to denote the weights and biases in the  $l^{\text{th}}$  layer. Further, the notation  $b_j^{(l)}$  refers to the  $j^{\text{th}}$  unit's bias in layer  $l$ , and  $w_{i,j}^{(l)}$  is the weight of the connection from unit  $i$  in layer  $l - 1$  to unit  $j$  in layer  $l$ . This notation follows Goodfellow *et al.* [1] and is customary in describing neural networks. We shall use  $n_0, n_1, \dots, n_L$  to denote the number of units in each layer. Layer zero is considered the *input layer*, meaning that  $n_0$  must be equal to the dimensionality of  $\mathbf{x}$ . The layers  $1, 2, \dots, L - 1$  are the *hidden layers* because they are not directly connected to the input or output units. Finally, we refer to layer  $L$  as the *output layer*, since its units represent the prediction  $\hat{\mathbf{y}}$ , and so  $n_L$  must be equal to the dimensionality of  $\hat{\mathbf{y}}$ . Figure 2.4 gives an example of a MLP architecture. It can clearly be seen that since MLPs are simply nested SLPs, they still form DAGs and thus are feedforward networks. In fact, such networks are usually *fully-connected feedforward networks* because all units from a particular layer connect to all units from the previous layer.

### 2.2.2 Backpropagation

*Training* with regard to neural networks refers to the process of altering a network's weights and biases with the goal of achieving an optimal configuration that reduces the error of the predictions, i.e. how far they are 'off'. This is how the network facilitates *learning* the input-output function from eq. (2.1) that we introduced at the very beginning of this chapter in the context of supervised learning.

Backpropagation with gradient descent is an iterative algorithm for training neural networks that, provided a suitable learning rate  $\alpha$ , is guaranteed to converge to a *local minimum*. The main idea is as follows:

1. Calculate the derivative of the loss function with respect to the current trainable parameters  $\mathbf{p}$  (weights and biases) as  $\Delta\mathbf{p} = \frac{\partial L}{\partial \mathbf{p}}(\mathbf{p})$ .
2. Take a step in the negative direction of this gradient, i.e. update the trainable parameters  $\mathbf{p} \leftarrow \mathbf{p} - \alpha \Delta\mathbf{p}$  for some learning rate  $\alpha \in \mathbb{R}$ .

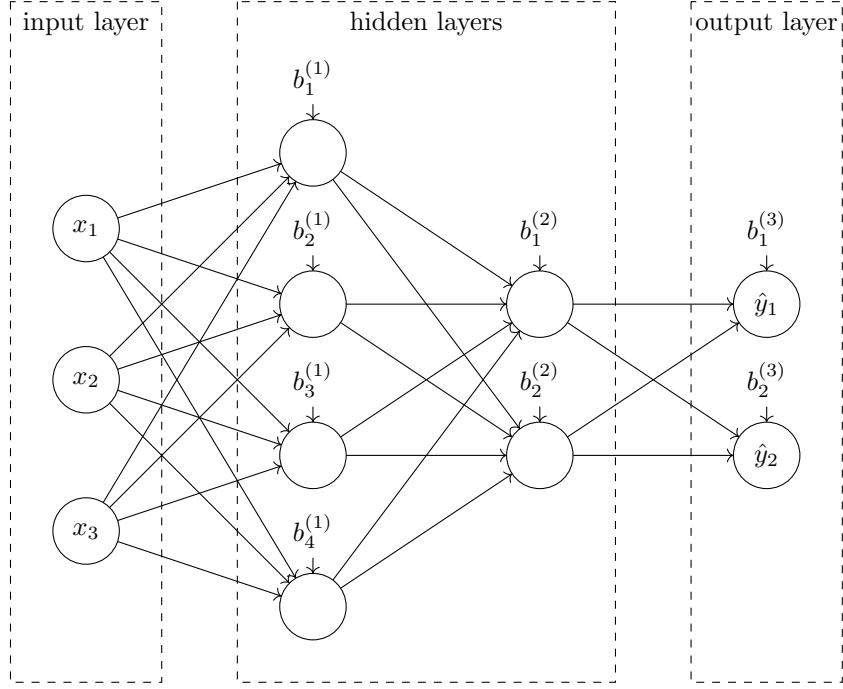


Figure 2.4: A MLP with three inputs, two hidden layers, and two outputs.

3. Repeat steps 1 and 2 until a predefined convergence criterion is met.

Figure 2.5 shows the steps that this algorithm would make on a simple error-weight surface with only one parameter.

To explain the backpropagation algorithm, we shall examine the case of a regression task, meaning that the network has only one output unit. The algorithm can, of course, be generalised to classification problems with multiple output units. Let  $\{\langle \mathbf{x}_i, y_i \rangle\}_{i=1}^m$  be a labelled regression dataset as defined in section 2.1.1.

We must first introduce a measure indicating how good our predictions are. For simplicity, we will use the mean squared error (MSE), although note that for classification problems, the cross-entropy loss is preferred. The MSE of a set of predictions  $\hat{\mathbf{y}}$  is the average squared difference between the predictions and targets,

$$E(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2 = \frac{1}{m} \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2. \quad (2.13)$$

The reader should not confuse this notation with that of a classification problem; here, we use  $\mathbf{y}$  to denote the targets for each sample in the dataset and  $\hat{\mathbf{y}}$  to denote the corresponding predictions.

We can formulate a loss function

$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2} \sum_{i=1}^m (\hat{y}_i - y_i)^2 \quad (2.14)$$

that behaves similarly to eq. (2.13) in that reducing the loss function will also reduce the MSE, but it will be easier to work with.

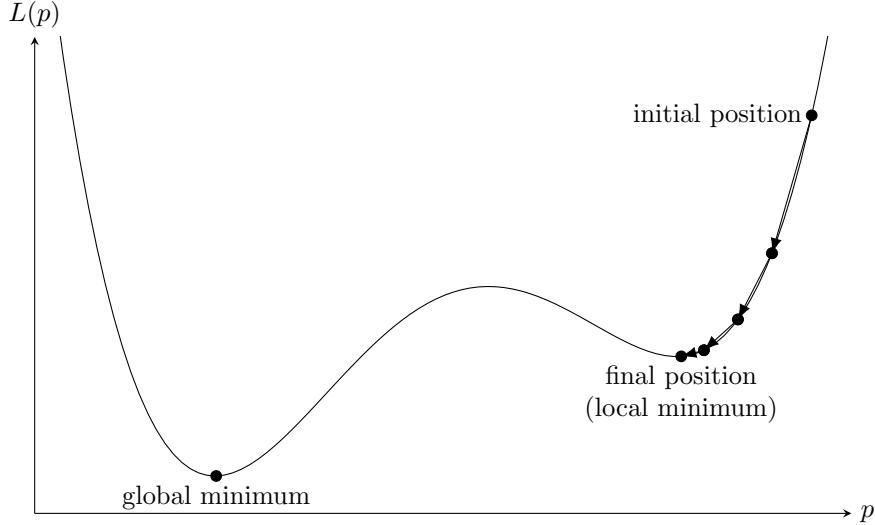


Figure 2.5: An illustration gradient descent training on an error-weight surface with only one parameter (not drawn to scale). Gradient descent minimises the objective loss function and converges to a local minimum. In practice, deep neural networks have parameters in the order of one million, resulting in a high-dimensional error surface, and local minima often turn out to be ‘good enough’ in that their error is only slightly higher than the global minimum [45].

### 2.2.2.1 Single-layer perceptron

In a SLP, we can easily express the loss in terms of the weights and biases by combining eqs. (2.11) and (2.14) as

$$L = \frac{1}{2} \sum_{i=1}^m (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i)^2. \quad (2.15)$$

Calculating the derivative of the loss function with respect to each of the trainable parameters is a core part of the gradient descent algorithm. The trainable parameters in our network are  $\mathbf{w}$  and  $b$ , so we will differentiate  $L$  with respect to each of these. We obtain the partial derivative of the loss with respect to the bias as

$$\begin{aligned} \frac{\partial L}{\partial b} &= \sum_{i=1}^m (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) \frac{\partial}{\partial b} (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) \\ &= \sum_{i=1}^m (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) g'(\mathbf{w}^\top \mathbf{x}_i + b), \end{aligned} \quad (2.16)$$

and similarly we can differentiate with respect to the weights

$$\begin{aligned} \frac{\partial L}{\partial \mathbf{w}} &= \sum_{i=1}^m (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) \frac{\partial}{\partial \mathbf{w}} (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) \\ &= \sum_{i=1}^m (g(\mathbf{w}^\top \mathbf{x}_i + b) - y_i) g'(\mathbf{w}^\top \mathbf{x}_i + b) \mathbf{x}_i. \end{aligned} \quad (2.17)$$

Here, the derivative  $g'(\cdot)$  depends on the choice of activation function  $g(\cdot)$ .

At each iteration, the gradient descent algorithm updates the trainable parameters by taking a step in the direction opposite to the gradient. The size of this step is governed by the learning rate  $\alpha \in \mathbb{R}$ . Mathematically, this is expressed as

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L}{\partial \mathbf{w}}(\mathbf{w}), \quad (2.18)$$

$$b \leftarrow b - \alpha \frac{\partial L}{\partial b}(b). \quad (2.19)$$

### 2.2.2.2 Multi-layer perceptron

To apply backpropagation in MLPs, we must first compute the gradient of the loss with respect to each trainable parameter (i.e. each layer's weights and biases). We must compute  $\frac{\partial L}{\partial \mathbf{W}_l}$  and  $\frac{\partial L}{\partial \mathbf{b}_l}$  for each layer  $l = 1, 2, \dots, L$ ; the exact derivation is left as an exercise to the reader. The main idea is realising that because MLPs are simply nested SLPs, we can apply the chain rule when calculating the derivative of eq. (2.12).

Similar to eqs. (2.18) and (2.19), once we have the gradients, we update the weights and biases by applying the rule

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \alpha \frac{\partial L}{\partial \mathbf{W}_l}(\mathbf{W}_l), \quad (2.20)$$

$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \alpha \frac{\partial L}{\partial \mathbf{b}_l}(\mathbf{b}_l). \quad (2.21)$$

A major limitation of gradient descent as we describe it here is the summation in eqs. (2.16) and (2.17) that is applied over the whole dataset of  $m$  samples. This becomes impractical for large datasets due to memory consumption and computational complexity, so instead we commonly split the dataset into batches and estimate the gradients for each batch in a process known as *stochastic gradient descent*. Furthermore, we typically apply more sophisticated optimisation algorithms than the update rules outlined in eqs. (2.20) and (2.21) to improve the speed and quality of convergence. A very popular choice that empirically demonstrates good results in many settings is the Adam optimiser [46]. However, such details are unfortunately outwith the scope of this report.

## 2.3 Convolutional neural networks

Let us examine how ANNs can be applied to a simple image classification problem. Consider an input image of  $256 \times 256$  pixels with three colour channels (RGB). If we flatten this image to a single vector, that vector will contain  $3 \cdot 2^{16}$  values. We could construct a MLP that takes this vector as input and tries to classify the image as belonging to one of the predefined classes. If the first layer has half as many units as the input, the weight matrix  $\mathbf{W}_1$  will be of size  $(3 \cdot 2^{16}) \times (3 \cdot 2^{15})$ , meaning that it will hold  $9 \cdot 2^{31}$  values. This means that the first layer alone will already have over two billion parameters. Networks with this many parameters are infeasible to train because the number of parameters exceeds the memory capacity and computational ability of modern

hardware. Furthermore, networks with too many parameters are much more likely to overfit to training data.

CNNs are a special class of ANNs that require significantly fewer parameters in each layer. They are well-suited for image tasks and can be trained on modern hardware even when they have many layers (which we refer to as *deep neural networks*). Originally, the concept of CNNs evolved from the so-called *neocognitron* that was introduced in 1980 and inspired studies of the visual cortex in humans and animals [47]. In 1998, LeCun *et al.* proposed the famous *LeNet-5* architecture that is considered to be the first CNN [48]. The main concepts developed in the seminal work of LeCun *et al.* carry through to today.

By conducting experiments on animals, David Hubel and Torsten Wiesel<sup>3</sup> found that biological neurons in the visual cortex respond to patterns in specific areas of the visual field, known as *receptive fields* [49]. Moreover, different neurons may respond to different patterns even if their receptive fields overlap. Hubel and Wiesel demonstrated this by showing that some neurons activate only for horizontal lines while others become active for vertical lines. They also showed that some neurons with larger receptive fields respond to more complex patterns that are combinations of such lower-level patterns.

CNNs follow the same intuition: neurons at the beginning of the network learn to recognise very simple, low-level patterns, while neurons deeper in the network compose these patterns to perceive more complex features. They are comprised of *convolutional* and *pooling* layers, which we shall describe below.

### 2.3.1 Convolutional layers

Convolutional layers are the most important component in a CNN. However, the term *convolution* is actually used as a misnomer to mean *cross-correlation* in the context of CNNs. Both terms refer to a mathematical operation where we slide a matrix (known as *filter* or *kernel*) over the input image and compute the sum of products at each location. A convolution operation is equivalent to cross-correlation if we pre-rotate the filter matrix by 180 degrees. For the remainder of this report we shall follow the convention in machine learning and use the term convolution to refer to cross-correlation.

Each neuron in the first convolutional layer is connected to a square region of pixels (the receptive field) from the input image, as opposed to being connected to the entire input image as is the case in fully-connected MLPs. Similarly, subsequent layers' neurons connect only to a square region of neurons in the previous layer, so that the network begins to form a hierarchy of filters.

Let us consider a neuron in the first convolutional layer. It is connected to a square patch of pixels from the input image (for now, we consider only one colour channel), so let us represent this patch using the  $k \times k$  matrix  $\mathbf{P}$  where  $k$  indicates the side length of the square. The convolutional layer is associated with a filter matrix  $\mathbf{F}$  of the same size as  $\mathbf{P}$ . The convolution of the matrices  $\mathbf{P}$  and  $\mathbf{F}$ , denoted  $\mathbf{P} * \mathbf{F}$ , is obtained by performing a pointwise multiplication of the corresponding elements in  $\mathbf{P}$  and  $\mathbf{F}$ , and then computing the sum of these results. More formally, using the notation  $p_{ij}$  and  $f_{ij}$  to index into those two

---

<sup>3</sup>Hubel and Wiesel received the Nobel Prize in Physiology or Medicine in 1981 for their work in understanding information processing in the visual system.

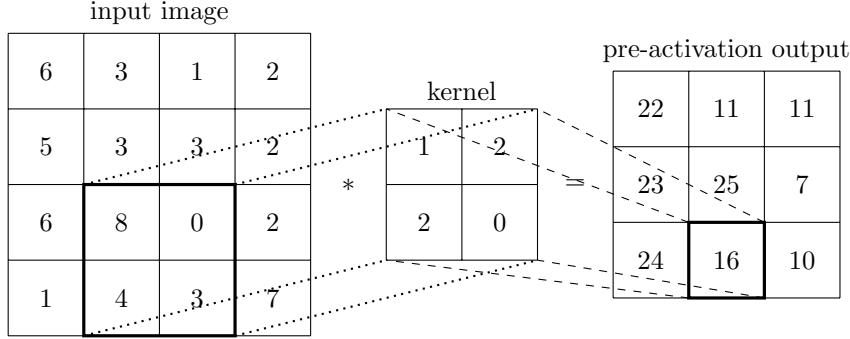


Figure 2.6: Illustration of a convolution operation with a  $2 \times 2$  kernel. The bias  $b$  will be added to the output on the right before the activation function  $g$  is applied pointwise in order to obtain the activations.

matrices, we can define the convolution operation as

$$\mathbf{P} * \mathbf{F} = \sum_{i=1}^k \sum_{j=1}^k p_{ij} f_{ij}. \quad (2.22)$$

A patch  $\mathbf{P}$  convolved with a particular filter  $\mathbf{F}$  will achieve a high value if the patch follows the pattern described by the filter. To see this, consider the filter

$$\mathbf{F} = \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

that tries to detect vertical lines. For a patch

$$\mathbf{P} = \begin{bmatrix} 1 & 8 & 2 \\ 2 & 9 & 2 \\ 1 & 11 & 1 \end{bmatrix}$$

that clearly has high values in the middle column and thus describes a vertical line, we have  $\mathbf{P} * \mathbf{F} = 47$ . However, if we consider a different patch

$$\mathbf{P}_2 = \begin{bmatrix} 10 & 11 & 10 \\ 9 & 11 & 10 \\ 10 & 10 & 9 \end{bmatrix}$$

where the intensities are quite uniform (and thus not describing a vertical line), the convolution with  $\mathbf{F}$  achieves a much lower result of 6.

Coming back to our examination of the first layer of a CNN, we know that it will be associated with a filter  $\mathbf{F}$ . The output of the layer is computed by sliding a  $k \times k$  window over the input image, and at each location of the window, we compute the convolution  $\mathbf{P} * \mathbf{F}$  where  $\mathbf{P}$  are the pixels in the current window. This process is illustrated using an example in fig. 2.6. The result will be a matrix, too, and so the output of the layer is obtained by applying the nonlinear activation function  $g$  pointwise to that matrix plus a bias  $b$ .

Now let us consider the case where the previous layer has multiple channels. The input image, for instance, has three colour channels. Thus, the first convolutional layer will need three filters: one for each input colour channel. The

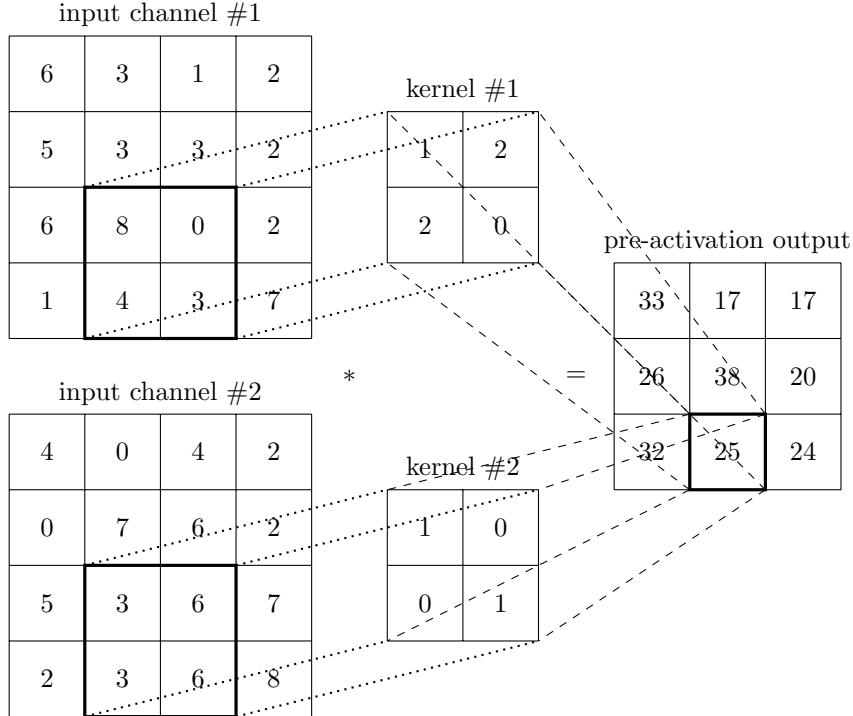


Figure 2.7: Illustration of a convolution operation with  $2 \times 2$  kernels acting on two input channels.

output is still only one matrix where the elements are summed across the channels, as explained in fig. 2.7 for a two-channel input. In practice, a convolutional layer will stack multiple such operations on top of each other (with different filters) so that we obtain more than one output matrix. These matrices, after adding the bias and applying the activation function  $g$ , will constitute the input channels for the subsequent layer. Each output channel will have its own bias  $b$ , so we require a bias vector  $\mathbf{b}$  to denote all of the biases.

The astute reader might wonder how we obtain the values for each of the filters so that they correspond to useful features in the image that will ultimately allow a series of fully-connected feedforward layers following the convolutional layers to perform a classification prediction at the end of the network. As in MLPs, the answer is to use backpropagation and gradient descent. Each of the values in the filter matrices is a trainable parameter and the biases are of course trainable, too. It is possible to calculate the gradients of these trainable parameters with respect to the loss function, so we can use an optimisation algorithm such as the popular *Adam* optimizer [46] to update the parameters as discussed at the end of section 2.2.2.2.

There are a number of hyperparameters associated with convolutional layers that impact the output dimensions as well as number of trainable parameters in the layer:

- the filter size  $k$ ;
- the padding  $p$  which indicates by how many rows/columns we extend the

input matrix in each direction, setting these new values to zero (figs. 2.6 and 2.7 use  $p = 0$ );

- the stride  $s$ , indicating how many pixels we move the sliding window each time (figs. 2.6 and 2.7 use  $s = 1$ ); and
- the desired number of output channels  $c_{\text{out}}$ .

Of course, the activation function  $g$  may also be regarded as a hyperparameter, but it has no effect on the size of the layer’s output. Let us consider a convolutional layer with input of spatial size  $h_{\text{in}} \times w_{\text{in}}$  with  $c_{\text{in}}$  channels. By examining how the spatial output dimensions correspond to the input size and hyperparameters in fig. 2.6, we can derive the equations

$$h_{\text{out}} = \left\lfloor \frac{h_{\text{in}} + 2p - k}{s} \right\rfloor + 1 \quad \text{and} \quad w_{\text{out}} = \left\lfloor \frac{w_{\text{in}} + 2p - k}{s} \right\rfloor + 1 \quad (2.23)$$

relating the input spatial dimensions to the output spatial dimensions of a convolutional layer. It is important to know this when designing a CNN architecture in order to monitor how the spatial size shrinks with each convolutional layer.

Having said that, we can finally provide evidence for the claim at the beginning of section section 2.3 that CNNs require significantly fewer trainable parameters than MLPs. Notice that the number of trainable parameters is simply the number of elements in each filter  $\mathbf{F}$  multiplied by the number of filters, plus the number of biases. For one output channel, we need  $c_{\text{in}}$  filters as shown in fig. 2.7. So, for  $c_{\text{out}}$  output channels, we require  $c_{\text{in}} \cdot c_{\text{out}}$  filters, and since each filter is a  $k \times k$  matrix, the number of kernel parameters is  $c_{\text{in}} \cdot c_{\text{out}} \cdot k^2$ . Each filter is associated with a bias that is trainable, so the total number of trainable parameters is  $c_{\text{in}} \cdot c_{\text{out}} \cdot (k^2 + 1)$ . In contrast to MLPs, this is independent of the spatial size of the input ( $h_{\text{in}} \times w_{\text{in}}$ ), and since the value of  $k$  is typically quite small, the number of trainable parameters will be orders of magnitude smaller than that of a layer in a MLP. As such, the weights in a CNN are considered to be *shared* because we slide the filters over the entire input and can therefore identify similar features in different spatial locations.

### 2.3.2 Pooling layers

To understand the motivation of pooling layers, we shall first examine the general architecture of a CNN. Typically, a CNN consists of a number of convolutional and pooling layers followed by a small MLP consisting of one or two fully-connected layers. Since the output of the convolutional/pooling layers is a tensor with three dimensions (more specifically, we have  $c_{\text{out}}$  feature maps, each of size  $h_{\text{out}} \times w_{\text{out}}$ ), it must first be flattened so that all elements are concatenated into a vector. However, to ensure that this vector contains useful information for classification, we would like to have a relatively small spatial size  $h_{\text{out}} \times w_{\text{out}}$ , but a large number of channels. The reasoning for this is two-fold: (i) a low spatial size will reduce the size of the vector and thus the number of trainable parameters in the MLP; and (ii) a high number of channels will allow the CNN to detect many different useful features. Pooling layers provide a means of reducing the spatial size while keeping the number of channels constant. They are used immediately after convolutional layers and typically occur less often than convolutional layers in CNN architectures because excessive use

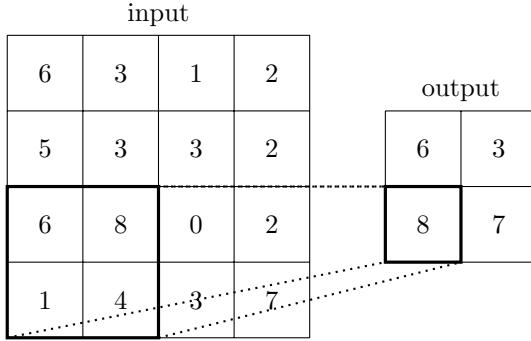


Figure 2.8: Illustration of maximum pooling for  $k = s = 2$ .

will prevent the network from learning any useful features. This is because pooling layers remove some information.

Pooling is applied on a per-channel basis, so the number of input channels is equal to the number of output channels. Furthermore, we do not apply an activation function because that is the job of the convolutional layers. The most common type of pooling is maximum pooling. In a max-pooling layer, we slide a window over the input and retain only the highest activation, as illustrated in fig. 2.8. It is easy to see that unlike convolutions, pooling layers have no trainable parameters. Nonetheless, they have three hyperparameters: the padding  $p$ , stride  $s$  and window size  $k$ . We often set  $s = k = 2$  and use no padding in order to halve the spatial dimensions (the arithmetic from eq. (2.23) holds for pooling layers as well).

## 2.4 Transfer learning

The underlying motivation behind transfer learning is to reuse knowledge gained in one task as a basis for learning a different task. It is an important tool that we employ in two different settings through this project: (i) training CNNs for classifying chess squares pieces in sections 4.2 and 4.3; and (ii) fine-tuning these models in order to adapt to new chess sets in chapter 5.

Training a deep CNN from scratch (starting from randomly initialised weights) can take several days even on modern GPUs. Furthermore, to achieve good performance, we require very large labelled datasets. Obtaining the required compute capability and quantity of data is often infeasible, and so it is very common to initialise the weights based on an already trained network. This makes sense because as discussed in section 2.3, CNNs learn patterns at the beginning of the network that are of a similar level of abstraction for different datasets. Understanding these feature representations is still a very active area of research; in fact, a recent paper shows that when training the same deep neural network architecture on different datasets, the learned features are very similar at the beginning of the network until a depth of around two thirds [50]. In practice, this means that it is only in the last third of the network that it specialises on the task at hand, and before that it learns only very general features.

For many popular deep CNN architectures, weight states are available online

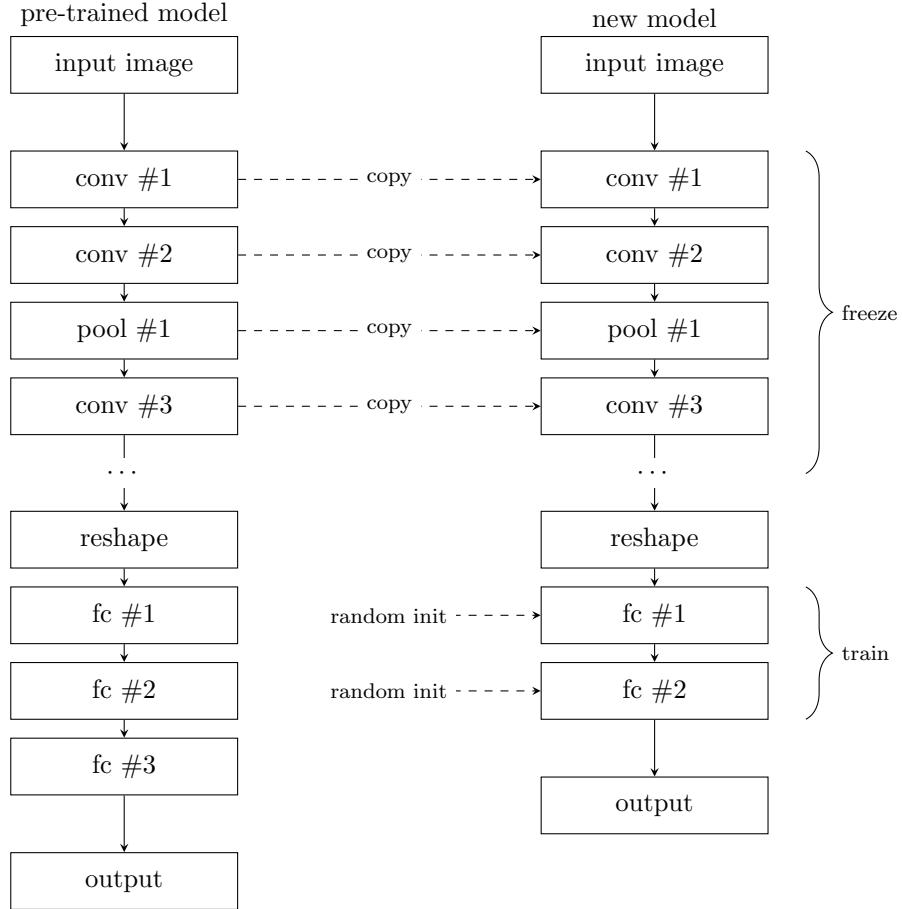


Figure 2.9: Illustration of the transfer learning process for domain adaption in deep neural networks. The boxes represent different types of layers: convolutional (conv), pooling (pool), and fully connected (fc). In the first instance, we train only the classification head, while the other layers are frozen. Then, we train the whole network.

that were trained for a long period of time on large datasets such as *ImageNet* [51], a classification dataset with millions of samples that are annotated for thousands of categories. Naturally, the *classification head* (the MLP after the convolutional and pooling layers that is responsible for classification) will need to be quite different for the target domain, especially if the target domain has a different number of classes. Thus, we will simply replace the classification head by a MLP that contains the desired number of output units, but use the pretrained weights and architecture for the preceding layers. Then, we can train the classification head in isolation by *frozening* the weights in all the other layers. In essence, this will train the final MLP based on the features learnt from the pretrained model. Finally, we can train the whole network by unfreezing all layers in a process called *fine-tuning*. Typically, this is performed with a smaller learning rate because the network will be significantly more sensitive. Figure 2.9 shows which layers are copied and trained using a simple example.

*I don't look at computers as opponents. For me, it is much more interesting to beat humans.*

Magnus Carlsen

# 3

## Data synthesis

Studies in human cognition by Bilalić *et al.* [52] and Zhou [53] compared skilled chess players with novices in terms of their ability to remember a chess position for a short amount of time and confirmed that highly skilled players outperform novices at this task. Perhaps more interestingly, both studies find that skilled players remember *random* positions (where pieces are positioned on random squares, not necessarily obeying the rules of chess) significantly less accurately than positions from actual chess games. Thus it stands to reason that in general, highly skilled chess players exhibit a more developed pattern recognition ability for chess positions than novices, but this ability is specific to positions that conform to the rules of chess and are likely to occur in actual games.

This project aims to develop a similar pattern recognition ability using machine learning, and therefore our dataset will consist of positions from real chess games. In doing so, we automatically ensure that the chess positions are legal and sensible.

### 3.1 Chess positions

The positions are generated from a publicly available dataset of 2,851 games played by current World Chess Champion Magnus Carlsen [54]. We randomly select 2% of all positions (i.e. configurations of chess pieces) from all games to be included in our dataset, although duplicate positions are discarded. A total of 4,888 chess positions are obtained in this manner and saved in FEN format.

### 3.2 Three-dimensional renders

In order to obtain realistic images of these chess positions, we employ a three-dimensional model of a chess set on a wooden table. Chess pieces are placed on the board squares according the given FEN description. Different camera

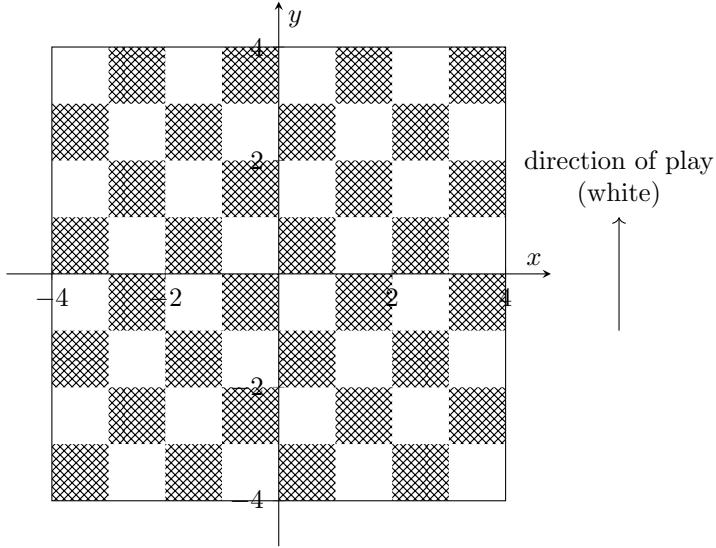


Figure 3.1: Overhead view of the coordinate system on the chessboard. The  $z$ -axis (not shown) points upward, normal to the chessboard surface, and the board is oriented like a chess game would be set up, i.e. the bottom right square is white. White's direction of play (the direction in which pawns are advanced) coincides with the  $y$ -axis.

angles and lighting setups are chosen in a random process in order to maximise diversity in the dataset.

Let us consider a three-dimensional Cartesian coordinate system whose origin lies at the centre point of the chessboard's surface, as depicted in fig. 3.1. The chessboard lies on the plane formed by the  $x$  and  $y$  axes, and the chess squares are of unit length.

**Pieces** The pieces are positioned on the squares as dictated by the particular FEN description. However, instead of positioning them at the centre in their respective squares, they are randomly rotated and positioned with a random offset to emulate the conditions in real chess games. More specifically, the  $x$  and  $y$  position of a piece in file  $i$  and rank  $j$  is sampled from a bivariate normal distribution given by

$$\mathbf{p}_{i,j} \sim \mathcal{N}\left(\begin{bmatrix} i \\ j \end{bmatrix} - \frac{7}{2}, \frac{\mathbf{I}_2}{10}\right).$$

Here, we assume that  $i$  and  $j$  are zero-indexed, i.e. the square  $a1$  corresponds to  $i = j = 0$ . The reason for shifting the mean by  $\frac{7}{2}$  above is that since the origin lies at the midpoint of the board, the mean must be shifted four units to the left (or downwards for the  $y$ -axis), but since the normal distribution should be centred on the midpoint of the square, we must add one half. Due to the fact that the  $x$  and  $y$  axes are perpendicular, the two components of  $\mathbf{p}$  will be independent and thus can be modelled with a covariance matrix that is a multiple of the identity matrix  $\mathbf{I}_2$ . Experiments showed that a variance of  $\frac{1}{10}$  achieved realistic results. Finally, the piece's rotation about its  $z$ -axis is sampled from a uniform distribution over the half-open interval  $[0, 2\pi)$ .

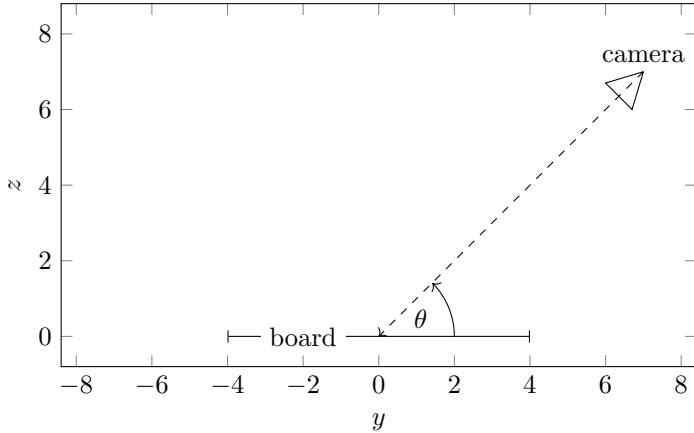


Figure 3.2: Side view of the camera setup for the scenario where it is white to move.

**Camera** The camera is aligned such that it points directly at the origin (i.e. the centre of the board). It is positioned with only a small offset from the  $yz$ -plane to ensure that the view over the chessboard is similar to the current player’s perspective. A slight perturbation to the  $x$ -component of the camera position is introduced according to a normal distribution with  $\mu = 0$  and  $\sigma = 0.8$  since the player will not usually be positioned exactly in the middle in front of the board. An angle  $\theta$  is chosen uniformly in the range  $[\frac{\pi}{4}, \frac{\pi}{3}]$  to represent the angle that the camera makes with the board’s surface (see fig. 3.2) if white is to play<sup>1</sup>. This range is chosen because human players would typically choose a camera angle between 45 and 60 degrees to ensure maximum visibility of the pieces. The two remaining components ( $y$  and  $z$ ) of the camera’s location are then obtained using a simple trigonometric calculation such that the distance from the camera to the origin is 11 units, a length that allows the camera to capture the entire board.

**Lighting** For each chess position, a random choice is made between two different lighting scenarios, each having equal probability of being employed.

1. The first lighting mode tries to emulate a *camera flash*. To do so, a spotlight is set up with the same location and orientation as the camera. As a result, the scene is lit up quite well with no large shadows, as it can be seen in fig. 3.3(a).
2. In the other lighting mode, two spotlights are set up in the scene. Their  $x$  and  $y$  coordinates are constrained such that they lie on a circle centred at the origin of the coordinate system with radius 10 on the  $xy$ -plane, as depicted in fig. 3.4, but each spotlight’s location along the circumference is sampled uniformly. Furthermore, each spotlight’s  $z$ -component is sampled uniformly in the range  $[5, 10]$ . Finally, for each spotlight, a focus point on the chessboard surface (i.e. the  $xy$  plane) is sampled from  $\mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \frac{5}{2}\mathbf{I}_2\right)$

<sup>1</sup>On the other hand, if it is black to play, the perspective must be from the other side of the board, so  $\theta$  is chosen in the range  $[\frac{2\pi}{3}, \frac{3\pi}{4}]$  which is equivalent to reflecting the camera position about the  $xz$ -plane.

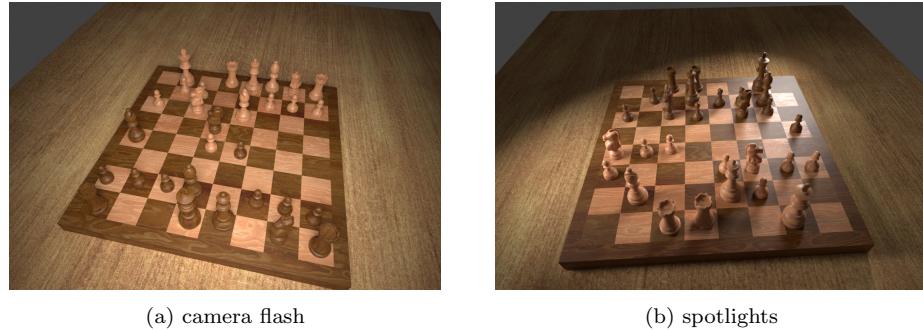


Figure 3.3: Two samples from the synthesised dataset showing both types of lighting.

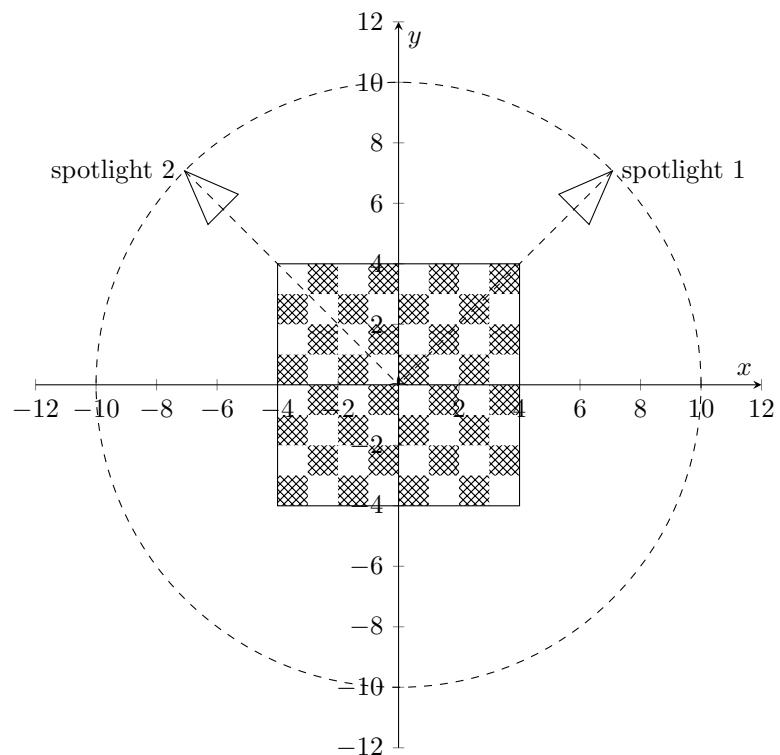


Figure 3.4: Overhead view of the chessboard with two spotlights. The spotlights are constrained to the dashed circle such that their distance to the origin amounts to 10 units when disregarding the  $z$ -component.

and the corresponding spotlight is rotated such that it points in that direction. Consequently, there is greater variability in the lighting because the spotlights might point at different areas on the board, thus producing different types of shadows. Figure 3.3(b) shows an example rendering where the lighting produced by the spotlights is poorer than the camera flash mode.

### 3.3 Automated labelling

Apart from eliminating the need to manually set up a large number of positions on the chessboard and photographing them, the use of a 3D model for synthetic data generation exhibits another major advantage: we can compute the required labels for the data without human intervention. In fact, manually labelling the data is usually the most time-consuming part of the entire process.

With each render, we export a file in JavaScript object notation (JSON) format alongside the image file containing the following information:

- the pixel coordinates of the chessboard’s four corner points on the rendered image;
- the FEN description of the pieces on the board;
- the colour of the current player (white/black);
- pixel coordinates of each piece’s bounding box on the rendered image;
- the position and angle of the camera (in the 3D world coordinate system); and
- the lighting mode as well as position, angle, and focus point of each light.

The coordinates of the chessboard’s corners can be computed because we know the coordinates of the relevant meshes in the 3D world coordinate system, and we know the world-to-camera-view matrix used to render the image. It is just a matter of performing some linear algebra to compute the corresponding 2D image coordinates. For computing the bounding boxes of the pieces, we simply project all vertices of the corresponding meshes onto the view plane and find the minimum and maximum points in the  $x$  and  $y$  directions. Figure 3.5 provides a visualisation of the automatically generated labels for one sample. Note that although we will not use the bounding boxes in this project, they are useful if someone should use the dataset to train an object recognition model for inferring chess positions instead of our classification-based approach. In this project, we will mainly rely on the board’s corner coordinates as well as the FEN description for training. Furthermore, the data about the cameras and lighting is useful in iteratively refining the models by analysing which conditions perform poorly and improving on them.

### 3.4 Splitting the dataset

In order to facilitate a fair evaluation of our chess recognition algorithm once it is completed, we immediately set aside 7% of the dataset as the so-called *test*



Figure 3.5: Visualisation of the automatically generated labels. For each piece, we compute the bounding box and annotate it with the piece type. The four corners of the chessboard are joined with red lines in the visualisation.

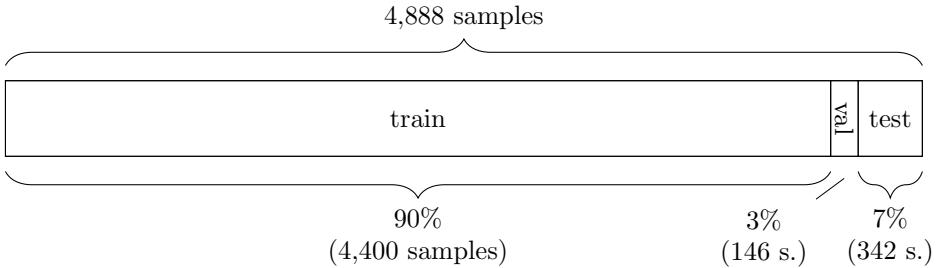


Figure 3.6: Visual representation of the dataset split.

set. We will not use this data until the very end when it is time to assess the performance of our chess recognition pipeline, so we can prevent *data leakage*: since we want to evaluate how our system performs on unseen data, we should not be allowed to use it during training. A much smaller portion of the data (3%) will be used for *validation* purposes. This subset will not directly be used for training, but for hyperparameter tuning or assessing whether a model overfits to the training data. Finally, the remaining 90% constitutes the training set. Figure 3.6 shows the proportions of the splits, including the actual number of samples in each subset. Note that the aforementioned splits are performed only once on the shuffled dataset and the associated JSON and image files are moved into three different folders to ensure that the same split is used each time.

*Of chess, it has been said that life  
is not long enough for it, but that  
is the fault of life, not chess.*

William Napier

# 4

## Recognising chess positions

The chess recognition system is responsible for taking an RGB image of a chessboard with pieces on it and ultimately produce a FEN string representing the predicted chess position. The pipeline, depicted in fig. 4.1, consists of four main stages which shall be described in detail in the subsequent sections.

The first step is locating the chessboard in the image. More specifically, we are interested in finding the pixel coordinates of the four chessboard corners. Using these corner points, we will warp the image such that the chessboard will form a regular square grid, to eliminate perspective distortion in the sizes of the chess squares. Next, we train a binary CNN classifier to determine individual squares' occupancies. Each of the occupied squares will then be input to another CNN that is responsible for determining the type of piece. Finally, we use the output probabilities of the piece classifier to produce a FEN string representing the position.

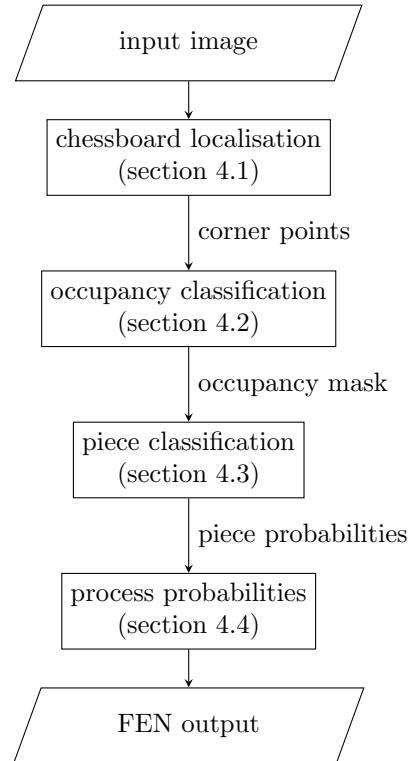


Figure 4.1: Overview of the chess recognition pipeline.

## 4.1 Board localisation

To determine the location of the chessboard's corners, we will rely on its regular geometric nature. Each square on the physical chessboard has the same width and height, even though their observed dimensions in the input image will vary due to 3D perspective distortion.

### 4.1.1 Finding intersection points

A chessboard consists of 64 squares that are arranged in an  $8 \times 8$  grid, thus there are nine horizontal and nine vertical lines. In the first step of our algorithm, we would like to detect the majority of these lines and find their intersection points. Before conducting any further computation, we resize the image to a width of 1,200 pixels, keeping the aspect ratio.

#### 4.1.1.1 Edge detection

To detect edges in the input image, we convert the RGB image to grayscale and apply the Canny edge detector [55], the result of which is shown in fig. 4.2(c). Canny edge detection is a multi-stage algorithm that first reduces noise in the image by applying a Gaussian blur, then calculates pixel intensity gradients, performs non-maximum suppression, and finally refines the results using hysteresis thresholding [55], although the precise details of this algorithm go beyond the scope of this report.

#### 4.1.1.2 Line detection

Next, we perform the so-called Hough transform [56], [57] in order to detect lines that are formed by the edges. To this end, we will represent lines in Hesse normal form, meaning that they are parameterised by the angle  $\theta$  that their normal vector makes to the  $x$ -axis, and the distance  $\rho$  to the origin. Figure 4.3 explains this geometrically and gives rise to the equation of a line in Hesse normal form,

$$\rho = x \cos \theta + y \sin \theta. \quad (4.1)$$

In fact, for a particular point  $(x, y)$ , eq. (4.1) represents the family of lines passing through it. Here, each pair of  $(\rho, \theta)$  values represents one particular line, and we only need to consider pairs where  $\rho \geq 0$  and  $0 \leq \theta < 2\pi$  to parameterise all lines.

Plotting the pairs of  $(\rho, \theta)$  values for a particular point in image space will give a sinusoidal curve that represents all lines passing through that point. We can take multiple points in image space and plot their sinusoids. Then, each intersection in  $(\rho, \theta)$  space will represent a line passing through the corresponding points, as illustrated in fig. 4.4.

For a given threshold  $t$ , the Hough transform essentially plots the sinusoids for all edge points in the input image (which we obtained using the Canny edge detection algorithm) and outputs intersection points in  $(\rho, \theta)$  space that are on at least  $t$  different sinusoids. Consequently, the lines identified by the Hough transform are corroborated by at least  $t$  edge points in the image.

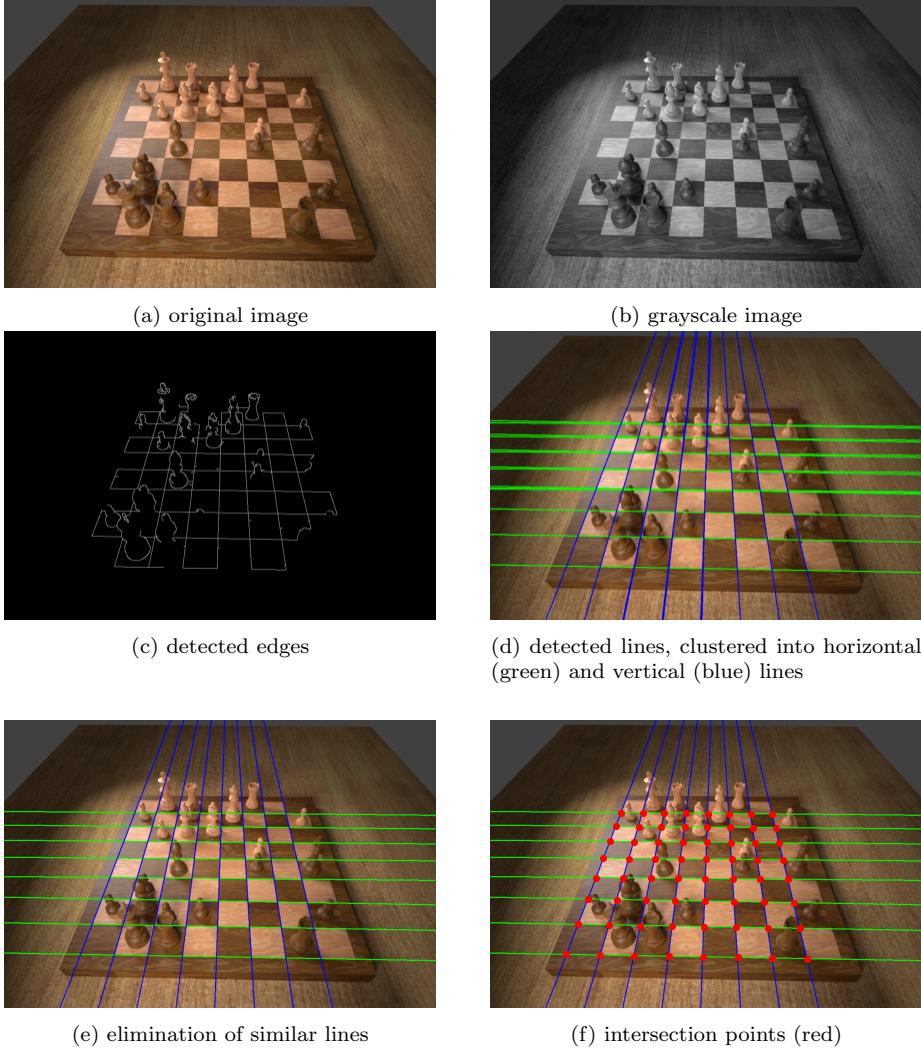


Figure 4.2: The process of determining the intersection points on the chessboard.

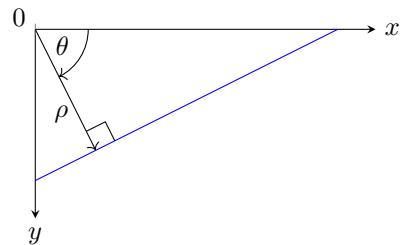


Figure 4.3: A line in Hesse normal form is parameterised by the angle  $\theta$  to the  $x$ -axis and distance  $\rho$  from the origin. The line is defined by the equation  $\rho = x \cos \theta + y \sin \theta$ . Equivalently, in slope-intercept form, we have  $y = -x \cot \theta + \frac{\rho}{\sin \theta}$  provided that  $\cos \theta \neq 0$ , i.e.  $\theta$  is not a multiple of  $\pi$ . Notice that the  $y$ -axis is pointing down because the origin of the coordinate system is the top left of the image.

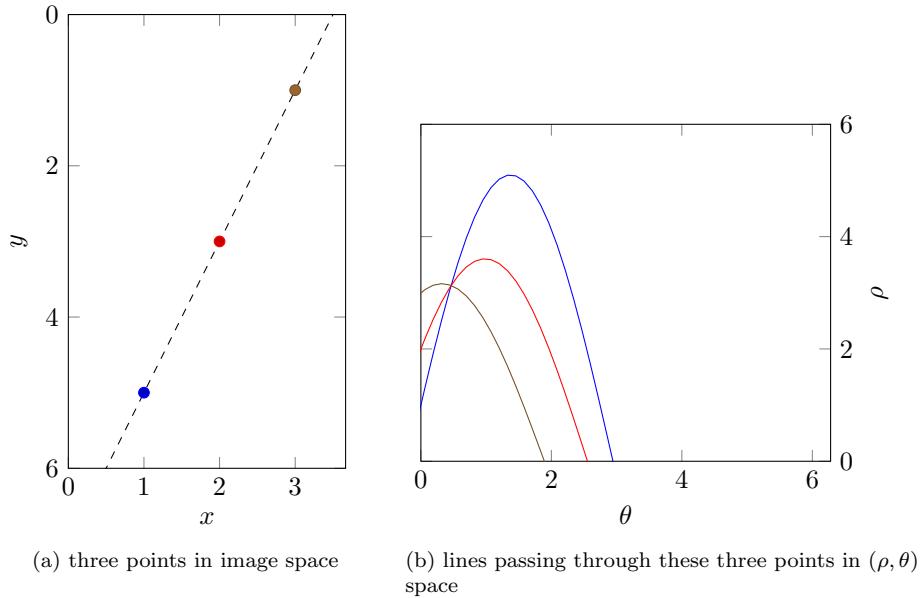


Figure 4.4: An example of how lines in image space are represented in  $(\rho, \theta)$  space. The family of lines passing through a particular point in image space (a) is represented by a sinusoid in  $(\rho, \theta)$  space (b). The intersection point of the three curves represents the line passing through all three points.

#### 4.1.1.3 Clustering

On the chessboard images, the Hough transform typically yields around 200 lines, most of which are very similar. We will first split them into horizontal and vertical lines and then eliminate similar lines. Experiments showed that simply setting thresholds for the angle  $\theta$  is insufficient for robustly classifying lines as horizontal or vertical. This is because the camera is often tilted quite severely in the synthetic dataset as a result of the procedure outlined in section 3.1 (and might be the case in real chessboard photos taken by humans, too). Instead, we employ agglomerative clustering which is a bottom-up hierarchical clustering algorithm. In this algorithm, each line starts off in its own cluster, and as the algorithm progresses, pairs of clusters are merged in a manner that aims to minimise the variance within the clusters. We cluster the lines based only on their direction, i.e. their angle  $\theta$ , and use the smallest angle between two given lines as the distance metric. Finally, we use the mean angle of both top-level clusters to determine which cluster represents the vertical lines and which the horizontal lines. The clustered lines are depicted in fig. 4.2(d).

Next, we must eliminate similar lines by finding clusters of similar lines. To eliminate similar horizontal lines, we first find the mean vertical line by considering the  $\rho$  and  $\theta$  values associated to the lines in the vertical cluster. Then, we find the intersection points of all the horizontal lines with the mean vertical line and perform a DBSCAN clustering [58] to group similar lines based on these intersection points. We use the mean  $\rho$  and  $\theta$  values of the lines in each group to represent the final set of discovered horizontal lines. The same procedure is applied vice-versa for the vertical lines, the result of which is shown

in fig. 4.2(e).

#### 4.1.1.4 Intersection points

It remains to find the intersection points of the horizontal and vertical lines. Given two lines parameterised by  $(\rho_1, \theta_1)$  and  $(\rho_2, \theta_2)$  respectively, we can find their point of intersection by observing that their equations in Hesse normal form (eq. (4.1)) constitute a system of two linear equations that can be solved for  $x$  and  $y$ . We have

$$\begin{aligned}\rho_1 &= x \cos \theta_1 + y \sin \theta_1 \\ \rho_2 &= x \cos \theta_2 + y \sin \theta_2.\end{aligned}$$

Rearranging for  $x$  and  $y$ , we obtain after some algebraic manipulation that

$$x = \frac{\rho_2 \sin \theta_1 - \rho_1 \sin \theta_2}{\cos \theta_2 \sin \theta_1 - \cos \theta_1 \sin \theta_2}, \quad (4.2)$$

$$y = \frac{\rho_2 \cos \theta_1 - \rho_1 \cos \theta_2}{\sin \theta_2 \cos \theta_1 - \sin \theta_1 \cos \theta_2}. \quad (4.3)$$

Finally, using eqs. (4.2) and (4.3) on each pair of horizontal and vertical lines, we can find the intersection points as depicted in fig. 4.2(f).

#### 4.1.2 Finding the homography

Once we have obtained the intersection points, the next step is to project them on to a regular grid. Using this projection, known as a *homography*, we can warp the original image to eliminate the perspective distortion.

##### 4.1.2.1 Computing the homography matrix

The projection is described by a *homography matrix*  $\mathbf{H} \in \mathbb{R}^{3 \times 3}$  [59] mapping any point  $\mathbf{p}$  from the original image to the corresponding point  $\mathbf{p}'$  in the warped image using the relation

$$\mathbf{H}\mathbf{p} = q\mathbf{p}' \quad (4.4)$$

up to a scalar factor  $q \in \mathbb{R}$ . Here, we consider  $\mathbf{p}$  and  $\mathbf{p}'$  as 2D *homogenous coordinate vectors*, which are three-component vectors that extend the Euclidean plane with points at infinity. More practically, a point  $(x, y)$  in Euclidean space corresponds to the set of homogenous coordinates  $[qx \quad qy \quad q]^\top$  for any  $q \in \mathbb{R}, q \neq 0$ . In practical terms, we can thus convert a homogenous coordinate to the corresponding Euclidean coordinate by taking its first two components and dividing by the third.

To compute a homography, we require four source points on the original image and four corresponding destination points in the warped image. We can choose four intersection points that lie on two distinct horizontal and two distinct vertical lines (according to the lines found in fig. 4.2(e)) and therefore represent a rectangle on the chessboard. Let the  $3 \times 4$  matrix  $\mathbf{P}$  contain these points' homogenous coordinates as column vectors in clockwise order. Let  $\mathbf{P}'$  be a matrix of the same size as  $\mathbf{P}$  containing this rectangle's coordinates in clockwise order. In the warped space, the squares of the chessboard will be of

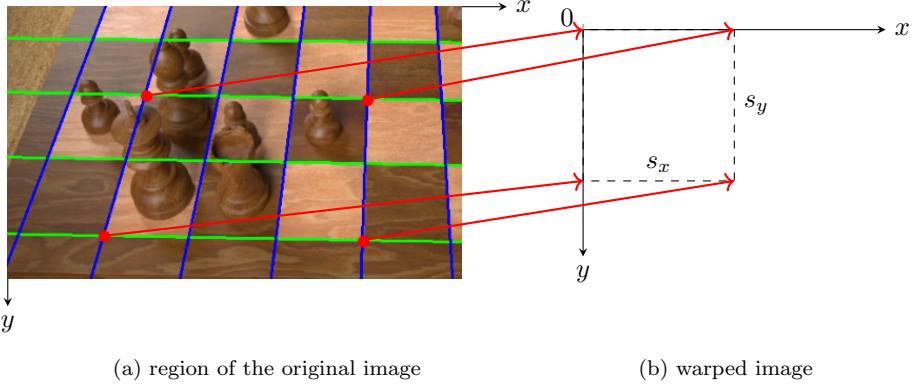


Figure 4.5: Projection of four intersection points from the original to the warped image.

unit length. Therefore, we can map the four points from the original image to a rectangle whose top left vertex coincides with the origin and whose side lengths are  $s_x$  and  $s_y$  (see fig. 4.5). We have

$$\mathbf{P}' = \begin{bmatrix} 0 & 0 & 1 \\ s_x & 0 & 1 \\ s_x & s_y & 1 \\ 0 & s_y & 1 \end{bmatrix}^\top.$$

Since the homography matrix maps points from the original image to the output image, we obtain from eq. (4.4) the relation

$$\mathbf{H}\mathbf{P} = \mathbf{P}'(\mathbf{I}_4\mathbf{q}) \quad (4.5)$$

where the vector  $\mathbf{q} \in \mathbb{R}^4$  represents the scale factor for each point. Equation (4.5) describes a system of linear equations that we can solve computationally for  $\mathbf{H}$ . Then, we can use eq. (4.4) to project all other intersection points to the warped image.

#### 4.1.2.2 Finding the optimal homography

Although fig. 4.2(f) has no intersection points that are outliers (because all of the identified lines do, in fact, correspond to squares on the chessboard), our algorithm must be robust when facing lines identified in the image that do not correspond to squares on the chessboard. Figure 4.6 depicts one such example; however, in non-synthetic chessboard images, especially with other objects in the image that exhibit straight lines, there is often an even greater number of outliers.

Random sample consensus (RANSAC) [60] is a non-deterministic iterative model fitting algorithm that can robustly estimate a model's parameters in such a manner that is not significantly influenced by outliers. It can be applied to a variety of situations where a model must be fit to a dataset containing outliers. In our case, the dataset consists of the intersection points (some of which are outliers due to detecting irrelevant lines) and the model is a homography that

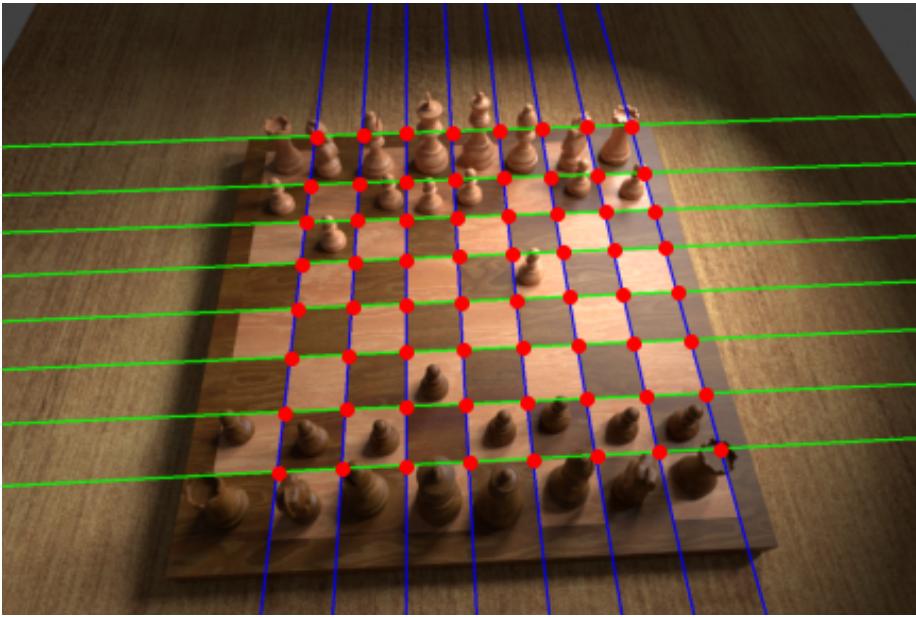


Figure 4.6: A sample from the training set with an incorrectly identified line. The top horizontal line is at the border of the chessboard instead of the top chessboard squares.

corresponds to the inlier intersection points. The RANSAC algorithm consists of two main steps:

1. Randomly sample a set of observations from the dataset that contains the minimum number of samples sufficient to fit the model. We require four points to compute a homography, so this is the number of intersection points we must sample in this step.
2. Fit the model to these samples, and then determine which observations are explained by the model within some threshold (the threshold should be just large enough to account for the expected noise in the dataset). This is the set of inliers, and if the size of this set is greater than the inliers discovered in the previous iteration, we will retain this model instead of the previous. Repeat from step 1 until reaching a specified number of iterations.

Applied to our problem, choosing any four intersection points in step 1 would be quite unwise because we would not know which points they represent on the chessboard, making it infeasible to map them to the warped image such that the chess squares are snapped to the grid of whole numbers. Instead, we randomly choose four points that we believe to form a rectangle on the chessboard. This means that we select two horizontal and two vertical lines at random, and choose the four intersection points of these lines as the random sample. We know that these four points will form a rectangle in the warped image, so we can map to a rectangle as explained in section 4.1.2.1 and illustrated in fig. 4.5.

The RANSAC-based method of finding the inlier intersection points is described in Algorithm 1. To simplify notation in the algorithm, we use the func-

---

**Algorithm 1** RANSAC-based algorithm for finding the optimal homography.

**Input:** the set of intersection points  $\mathcal{P} \subset \mathbb{R}^2$

**Parameters:** minimum number of iterations  $i_{\min}$ , maximum number of iterations  $i_{\max}$ , minimum number of inliers  $c$ , inlier threshold  $\epsilon_{\max}$

**Output:** a bijective mapping  $m : \mathcal{I} \rightarrow \mathcal{W}$  from a set of inlier points  $\mathcal{I} \subseteq \mathcal{P}$  to a set of warped points  $\mathcal{W} \subset \mathbb{Z}^2$  that is given by its graph  $\mathcal{M} \subseteq \mathcal{I} \times \mathcal{W}$

```

1:  $\mathcal{P} \leftarrow \{h(\mathbf{p}) : \mathbf{p} \in \mathcal{P}\}$                                  $\triangleright$  convert to homogenous coordinates
2:  $\mathcal{I}, \mathcal{W}, \mathcal{M} \leftarrow \emptyset, \emptyset, \emptyset$            $\triangleright$  initialise outputs
3:  $i \leftarrow 0$                                           $\triangleright$  number of iterations
4: while  $(i < i_{\min}) \vee (|\mathcal{I}| < c)$  do
5:    $x_1, x_2 \leftarrow \text{choose}(\{1, 2, \dots, x_{\max}\})$   $\triangleright$  randomly choose such that  $x_1 < x_2$ 
6:    $y_1, y_2 \leftarrow \text{choose}(\{1, 2, \dots, y_{\max}\})$   $\triangleright$  randomly choose such that  $y_1 < y_2$ 
7:    $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \mathbf{p}_4 \leftarrow$  intersection points in  $\mathcal{P}$  corresponding to the four points
       described by vertical lines  $x_1, x_2$  and horizontal lines  $y_1, y_2$  (in clockwise
       order)
8:    $\mathbf{P} \leftarrow [\mathbf{p}_1 \ \mathbf{p}_2 \ \mathbf{p}_3 \ \mathbf{p}_4]^{\top}$                                  $\triangleright$  matrix of original points
9:   for all  $s_x \in \{1, 2, \dots, 8\}$  do
10:    for all  $s_y \in \{1, 2, \dots, 8\}$  do
11:       $\mathbf{P}' \leftarrow \begin{bmatrix} 0 & 0 & 1 \\ s_x & 0 & 1 \\ s_x & s_y & 1 \\ 0 & s_y & 1 \end{bmatrix}^{\top}$            $\triangleright$  matrix of warped points
12:       $\mathbf{H} \leftarrow$  homography matrix calculated for  $\mathbf{P}$  and  $\mathbf{P}'$  using eq. (4.5)
13:       $\mathcal{I}_C \leftarrow \emptyset$            $\triangleright$  candidate set of inliers for this iteration
14:       $\mathcal{W}_C \leftarrow \emptyset$            $\triangleright$  candidate set of warped points
15:       $\mathcal{M}_C \leftarrow \emptyset$            $\triangleright$  candidate mapping
16:      for all  $\mathbf{p} \in \mathcal{P}$  do           $\triangleright$  iterate over all intersection points
17:         $\mathbf{p}' \leftarrow h^{-1}(\mathbf{H}\mathbf{p})$            $\triangleright$  warp the point according to our model
18:         $\mathbf{q} \leftarrow \text{round}(\mathbf{p}')$            $\triangleright$  snap to grid (round element-wise to nearest
       whole number)
19:        if  $\mathbf{q} \in \mathcal{W}_C$  then           $\triangleright$  ignore this point if we already mapped it
           continue
20:        end if
21:        end if
22:         $\epsilon \leftarrow \|\mathbf{p}' - \mathbf{q}\|_1$            $\triangleright$  calculate the error
23:        if  $\epsilon \leq \epsilon_{\max}$  then           $\triangleright$  determine whether we have an inlier
24:           $\mathbf{p} \leftarrow h^{-1}(\mathbf{p})$ 
25:           $\mathcal{I}_C \leftarrow \mathcal{I}_C \cup \{\mathbf{p}\}$            $\triangleright$  add the inlier to our candidate mapping
26:           $\mathcal{W}_C \leftarrow \mathcal{W}_C \cup \{\mathbf{q}\}$ 
27:           $\mathcal{M}_C \leftarrow \mathcal{M}_C \cup \{(\mathbf{p}, \mathbf{q})\}$ 
28:        end if
29:      end for
30:      if  $|\mathcal{I}_C| > |\mathcal{I}|$  then           $\triangleright$  update the set of inliers if we found more
            $\mathcal{I}, \mathcal{W}, \mathcal{M} \leftarrow \mathcal{I}_C, \mathcal{W}_C, \mathcal{M}_C$ 
31:      end if
32:    end for
33:  end for
34: end for
35: if  $i \geq i_{\max}$  then break end if           $\triangleright$  guard against iterating indefinitely
36:  $i \leftarrow i + 1$ 
37: end while

```

---

tion  $h : \mathbb{R}^2 \rightarrow \mathbb{R}^3$  to convert Euclidean coordinates to homogenous coordinates and  $h^{-1} : \mathbb{R}^3 \rightarrow \mathbb{R}^2$  to denote the reverse operation. Based on our characterisation of homogenous coordinates from section 4.1.2.1, these functions can be defined as

$$h(x, y) = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and

$$h^{-1}(x, y, z) = \begin{bmatrix} x/z \\ y/z \end{bmatrix}$$

provided that  $z \neq 0$ .

The input to algorithm 1 is the set of intersection points  $\mathcal{P} \subset \mathbb{R}^2$  in the original image. Its output is a bijection  $m : \mathcal{I} \rightarrow \mathcal{W}$  from a set of inlier points  $\mathcal{I} \subseteq \mathcal{P}$  to a set of warped points  $\mathcal{W} \subset \mathbb{Z}^2$ . Notice that the warped points are represented by two-dimensional vectors whose components are whole numbers, meaning that they lie on a grid of whole numbers. Each square on the chessboard is represented by a square of unit length on the grid. The chessboard itself is represented by a  $8 \times 8$  square on that grid. However, since the set of inliers will likely not contain all vertices of the chessboard squares (for example because some lines were not recognised as seen in figs. 4.2(e) and 4.6) we do not yet know the location of that  $8 \times 8$  square.

Once we have computed the output of algorithm 1 and obtained the mapping  $m$  of points from the original image to points on the grid of the warped image, we can recompute the homography matrix  $\mathbf{H}$  using eq. (4.5) with *all* inliers to obtain a more accurate homography that is based on more than just four points. To do so, we take the matrix of original points  $\mathbf{P}$  to be a  $3 \times |\mathcal{I}|$  matrix whose columns are populated with all the intersection points  $\mathcal{I}$  as homogenous coordinate vectors. The matrix of warped points  $\mathbf{P}'$  of the same size will be populated by the corresponding points in warped space  $\mathcal{W}$  according to the bijection  $m$ . Using the new  $\mathbf{P}$  and  $\mathbf{P}'$  matrices, eq. (4.5) becomes an over-determined linear system, so we compute  $\mathbf{H}$  as the least squared error solution instead of the exact solution. Finally, we can use the homography matrix to project the pixels of the original image to obtain a warped image as in fig. 4.7.

#### 4.1.2.3 Optimisations

Algorithm 1 aims to outline the method of determining the inlier points in a manner that is easy to follow. In practise, the employed algorithm will be different in order to achieve improved performance. Implementing these differences in algorithm 1 would have made it too convoluted to follow, so they shall instead briefly be summarised below:

- The for loop on line 16 can be vectorised and replaced by matrix operations.
- The computation of the homography matrix on line 12 occurs 64 times per iteration of the outer loop, due to the two for loops on lines 9 and 10. Instead, it should be moved before the two aforementioned for loops and computed for the scale  $s_x = s_y = 1$ . To ensure the points are mapped correctly, we must add the instruction  $\mathbf{p}' \leftarrow \mathbf{p}' \odot [s_x \ s_y]^\top$  immediately after line 17 in order to scale the warped points.

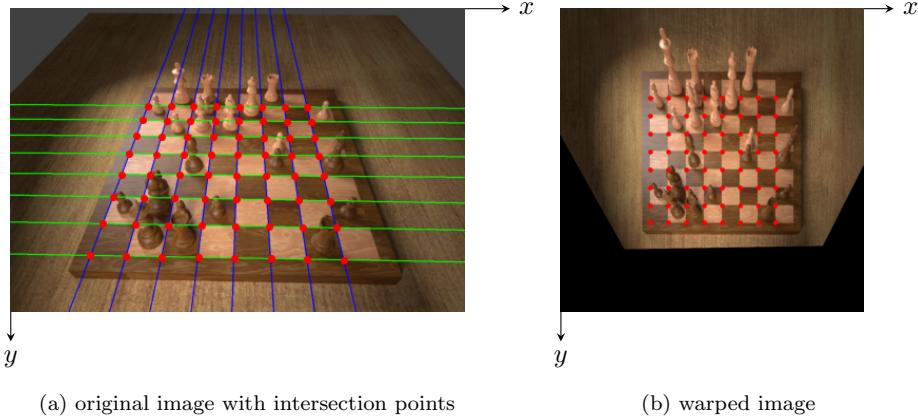


Figure 4.7: The original image is warped using the computed homography matrix  $\mathbf{H}$ . Detected lines and intersection points are overlaid in the original image as per fig. 4.2(e). The corresponding intersection points are marked in red in the warped image and lie on a regular unit-length grid.

- The for loops on lines 9 and 10 do not need to be nested; instead, one could first consider only the  $x$ -component of the vectors when calculating the errors for each scale  $s_x$ , and then do the same for the  $y$ -components and  $s_y$ . Finally, we would choose the best  $s_x$  and  $s_y$  (with the least accumulated error), compute the total error, and use the information to update the mappings if necessary. This is the reason for using the  $\ell_1$  norm for computing the error on line 22, since it can be easily decomposed into the two directional components.

#### 4.1.3 Inferring missing lines

It was noted at the end of section 4.1.2.2 that while the result of algorithm 1 projects the inlier intersection points onto a regular grid of whole numbers, it does not explicitly indicate the location of the  $8 \times 8$  sub-grid that represents the chessboard. To understand the set of inliers, we shall first compute the minimum and maximum  $x$  and  $y$  values of the warped points. We shall use the notation

$$x_{\min} = \min \mathcal{W}_x, \quad x_{\max} = \max \mathcal{W}_x, \quad (4.6)$$

$$y_{\min} = \min \mathcal{W}_y, \quad y_{\max} = \max \mathcal{W}_y, \quad (4.7)$$

where

$$\mathcal{W}_x = \left\{ x : [x \ y]^\top \in \mathcal{W} \right\} \quad \text{and} \quad \mathcal{W}_y = \left\{ y : [x \ y]^\top \in \mathcal{W} \right\}. \quad (4.8)$$

The  $x_{\min}$  value describes the leftmost vertical line in the warped image given by the equation  $x = x_{\min}$ , and the value of  $x_{\max}$  describes the rightmost vertical line. Similarly, the values of  $y_{\min}$  and  $y_{\max}$  represent the top and bottom lines. Examining the range of  $x$ -values ( $x_{\max} - x_{\min}$ ) gives an indication of whether the vertical lines we identified represent the whole chessboard, part of it, or even a range that is wider than the chessboard itself, horizontally. We thus distinguish between three cases based on the  $x$ -values which we shall outline below.

*Case 1.*  $x_{\max} - x_{\min} = 8$ .

In this case, no action is required because we detected the two outer vertical lines. The leftmost vertical line of the chessboard is located at  $x = x_{\min}$  and the rightmost line is at  $x = x_{\max}$  on the warped image.

*Case 2.*  $x_{\max} - x_{\min} > 8$ .

This case arises when the vertical line  $x = x_{\min}$  is to the left of the left edge of the chessboard and/or the vertical line  $x = x_{\max}$  is right of the right edge of the chessboard. In other words, the range of  $x$ -values covers more than just the chessboard. To remedy this, we will iteratively update  $\mathcal{W}_x$  by applying

$$\mathcal{W}_x \leftarrow \mathcal{W}_x \setminus \{x_{\min}, x_{\max}\}$$

and recomputing  $x_{\min}$  and  $x_{\max}$  by eq. (4.6) until  $x_{\max} - x_{\min} \leq 8$ . Then, we update the set of warped points  $\mathcal{W}$  as

$$\mathcal{W} = \left\{ [x \ y]^{\top} : [x \ y]^{\top} \in \mathcal{W}, x \in \mathcal{W}_x \right\},$$

retaining only the warped points whose  $x$ -component is in the interval  $[x_{\min}, x_{\max}]$ . We also remove the corresponding points from  $\mathcal{I}$  and the mapping  $m$ 's graph  $\mathcal{M}$ . Then, we fall either in case 1 or 3, depending on the values of  $x_{\min}$  and  $x_{\max}$ .

*Case 3.*  $x_{\max} - x_{\min} < 8$ .

This most common scenario arises when the grid that should represent the chessboard in the warped image is smaller than eight units in width. Figure 4.7 illustrates this case and makes clear that it arises because either one or both of the outermost vertical lines were not detected on the chessboard. Determining whether the grid should be expanded towards the left or the right involves further processing of the image which will be outlined below.

We will calculate the horizontal intensity gradients in the warped image for each pixel in order to determine whether an additional vertical line is more likely to occur one unit to the left or one unit to the right of the currently identified grid. To do so, we first convert the RGB image to grayscale and then apply the horizontal Sobel filter which is a discrete differentiation operator that provides an approximation for the gradient intensity in the horizontal direction. Large horizontal gradient intensities will give rise to vertical lines in the warped image which can be used in order to determine whether the chessboard should be expanded to the left or right.

The Sobel filter is widely used in image processing and consists of two kernels that are convolved over the input image, a horizontal and a vertical one. Typically, the results of both convolution operations are calculated, and then some elementary trigonometry can be applied in order to determine the gradient directions for each pixel. However, since we are only interested in determining the gradients in the horizontal direction, it suffices to use the horizontal Sobel kernel

$$\mathbf{S}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} \quad (4.9)$$

Given an intensity image (i.e. a grayscale image) of dimensions  $h \times w$  that is represented by the matrix  $\mathbf{A} \in \mathbb{R}^{h \times w}$ , we compute the horizontal gradient intensity at each pixel by

$$\mathbf{G}_x = \mathbf{S}_x * \mathbf{A} \quad (4.10)$$

where  $\mathbf{G}_x$  is of the same dimensionality as  $\mathbf{A}$  and contains the computed gradient intensities. Then, we can apply the Canny edge detector (as we did in section 4.1.1) to image represented by  $\mathbf{G}_x$  in order to eliminate noise and obtain clear edges. Figure 4.8 shows the results of these operations on the running example.

On that image we shall sum the pixel intensities on the vertical lines at  $x = x_{\min} - 1$  and  $x = x_{\max} + 1$  (with a small tolerance to the left and to the right). Then, if the sum of pixel intensities was greater at  $x = x_{\min} - 1$  than  $x = x_{\max} + 1$ , we will update  $x_{\min} \leftarrow x_{\min} - 1$ , or otherwise  $x_{\max} \leftarrow x_{\max} + 1$ . We repeat this process until  $x_{\max} - x_{\min} = 8$ .

Although the three cases above examine only the  $x$ -values to locate the horizontal position of the chessboard, it is trivial to see how this method applies to determine the vertical position of the chessboard using the  $y$ -values, too. However, the main caveat the reader should acknowledge is that we must use the vertical sobel kernel

$$\mathbf{S}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}. \quad (4.11)$$

instead of the horizontal one given in eq. (4.9), and as a consequence eq. (4.10) becomes

$$\mathbf{G}_y = \mathbf{S}_y * \mathbf{A}. \quad (4.12)$$

By following the instructions according to the three cases above for both the horizontal and vertical directions, we obtain the  $x_{\min}$ ,  $x_{\max}$ ,  $y_{\min}$ , and  $y_{\max}$  values that describe the location of the chessboard in the warped image. As such, the coordinates of the four corners of the chessboard on the warped image can be expressed as

$$\begin{aligned} \mathbf{p}'_1 &= [x_{\min} \ y_{\min}]^\top & \mathbf{p}'_2 &= [x_{\max} \ y_{\min}]^\top \\ \mathbf{p}'_3 &= [x_{\max} \ y_{\max}]^\top & \mathbf{p}'_4 &= [x_{\min} \ y_{\max}]^\top. \end{aligned}$$

The points are supplied in clockwise order, starting from the top left.

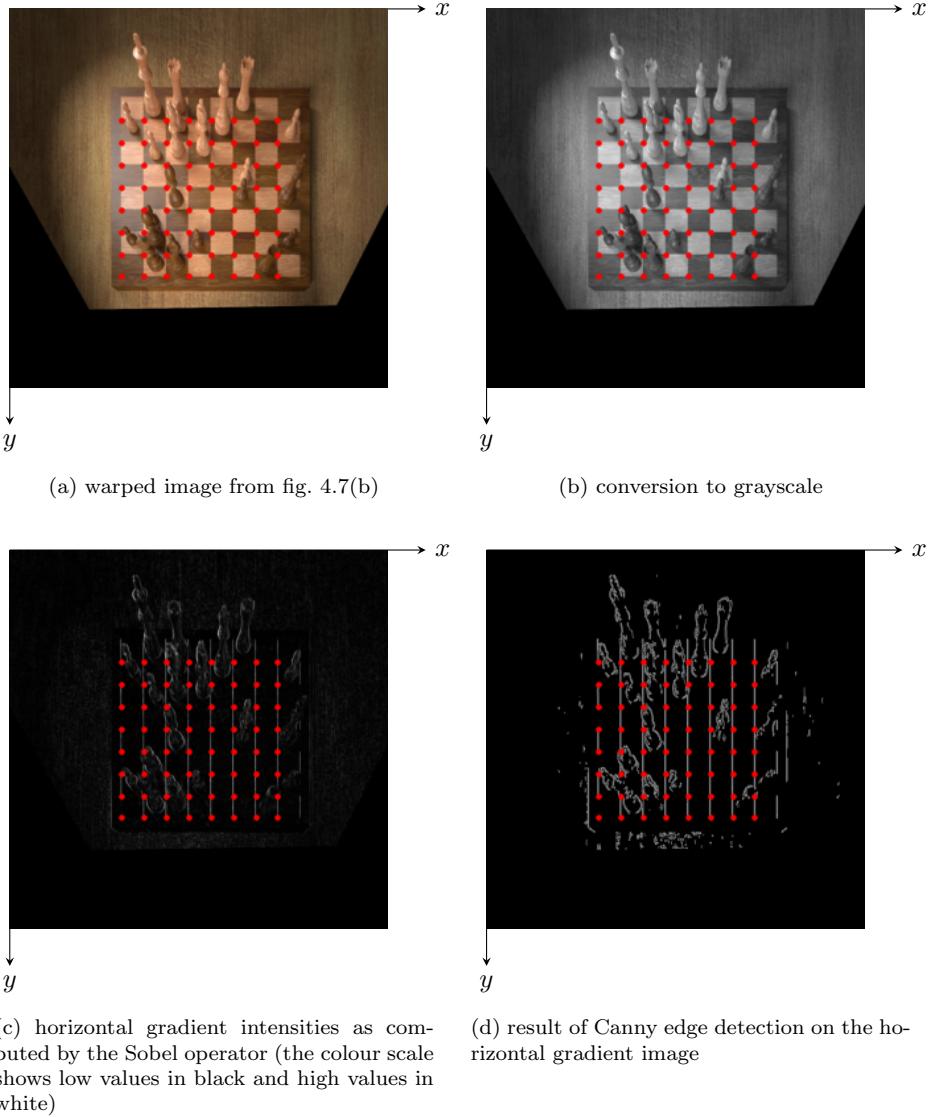
Finally, we can use the inverse of the homography matrix in order to project these points back onto the original image by inverting eq. (4.4). Notice that the points must be converted to homogenous coordinates and back again, so the equation becomes

$$\mathbf{p}_i = h^{-1}(\mathbf{H}^{-1}h(\mathbf{p}'_i)) \quad (4.13)$$

for  $i = 1, 2, 3, 4$ . The result is depicted in fig. 4.9.

#### 4.1.4 Determining optimal parameters

The method of finding the four corner points of the chessboard described in the previous sections relies on numerous parameters whose optimal values must still be determined. The most important parameters are summarised below:



(c) horizontal gradient intensities as computed by the Sobel operator (the colour scale shows low values in black and high values in white)

(d) result of Canny edge detection on the horizontal gradient image

Figure 4.8: Horizontal gradient intensities calculated on the warped image in order to detect vertical lines. The red dots correspond to the intersection points in  $\mathcal{I}$ . Here,  $x_{\max} - x_{\min} = 7$  because there are eight columns of points instead of nine. This is because the rightmost vertical line was not detected.

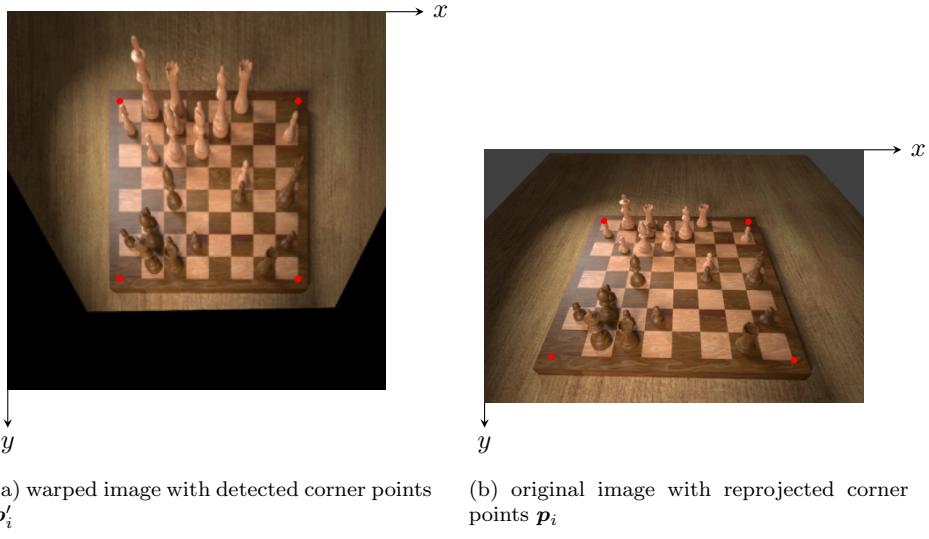


Figure 4.9: The identified corner points are projected back from the warped image onto the original image using the inverse homography matrix.

- low and high thresholds of the final stage of the Canny edge detector (hysteresis);
- threshold  $t$  of the Hough transform as explained in section 4.1.1;
- minimum number of iterations  $i_{\min}$ , maximum number of iterations  $i_{\max}$ , minimum number of inliers  $c$ , and the inlier threshold  $\epsilon_{\max}$  from algorithm 1;
- low and high Canny edge detection thresholds for the horizontal and vertical gradient images, as well as the relative width of the horizontal and vertical lines from section 4.1.3.

The optimal parameter values are decided by performing a grid search using sensible presets. A corner is classified as being detected accurately if the prediction is within ten pixels of the groundtruth label, which given the fact that the width of the image is 1,200 pixels ensures that the predictions must be very close to the groundtruth. If at least one corner is detected inaccurately, that particular sample is determined to be inaccurate. We perform the grid search only on a small subset of the training set because the number of different parameter sets to be tested is in the order of a thousand. The set of parameters achieving the best performance on this subset of the training set is retained as the final configuration.

Using this configuration, the corner detection algorithm yields inaccurate predictions in 13 cases out of 4,400 samples in the training set, so its accuracy is 99.71%. The validation set of size 146 sees no mistakes and thus achieves an accuracy of 100.00%. As such, we conclude that the corner detection algorithm is very robust to unseen samples from the dataset and did not overfit as a result of the grid search.

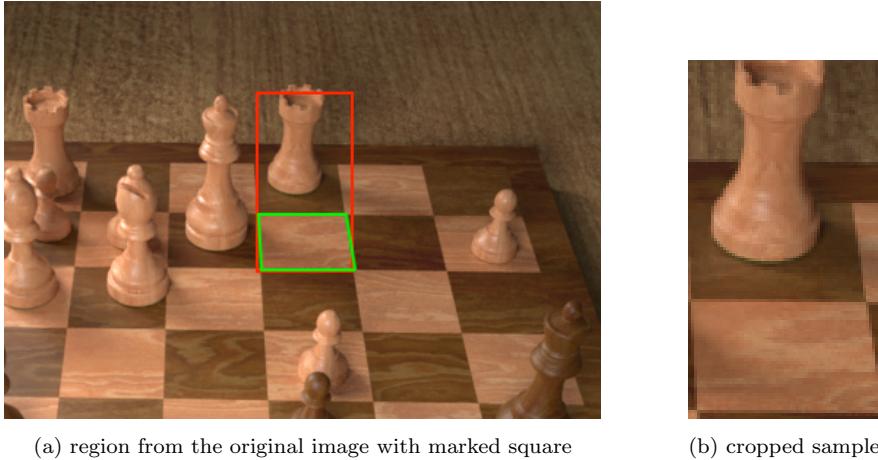


Figure 4.10: An example illustrating why an immediate piece classification approach is inclined to reporting false positives. Consider the square marked in green in the original image (a). The bounding box for piece classification (marked in red) must be quite tall because the square might contain a tall piece such as a queen or king (the box must be at least as tall as the the queen in the adjacent square on the left). The resulting sample, depicted in (b), contains almost the entire rook of the square behind. Thus, a piece classifier might classify this square as containing a rook instead of being empty.

## 4.2 Occupancy classification

Empirical experiments showed that performing piece classification directly after detecting the four corner points with no intermediate step yields a large number of false positives, i.e. empty squares being classified as containing a chess piece. One common scenario where the trained classifier failed is illustrated in fig. 4.10. Notice that squares further away from the camera must be cropped with increasingly taller bounding boxes. If a particular square is empty but its bounding box includes the piece from the adjacent square as in fig. 4.10(b), the trained classifier was inclined to report a false positive.

To solve this problem, a binary classifier is trained on cropped squares to decide whether they are empty or not. Before cutting out the squares from the original image, the input image is warped to a two-dimensional overhead view by means of the projective transformation outlined in section 4.1.2. This ensures that all squares are of equal size and that the corners form right angles (which is not the case in the original image due to perspective distortion), as depicted in figs. 4.11(a) and 4.11(b).

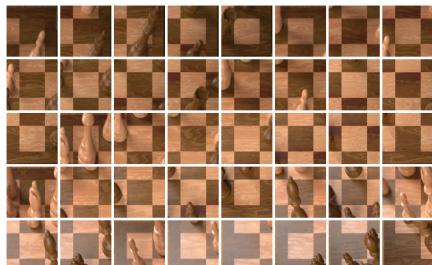
Cropping the squares from the warped image is trivial because the squares are of equal size. In training the occupancy classifiers, we hypothesise that it would be useful to include contextual information with each square; therefore, the squares are not cropped tightly around their boundaries but instead with a 50% increase in length on all four sides, as shown in figs. 4.11(c) and 4.11(d). This might aid the classifier's decision in difficult situations where a chess piece from another square reaches into the cropped one due to the camera perspective.



(a) original



(b) warped



(c) all 32 empty samples



(d) all 24 occupied samples

Figure 4.11: The process of obtaining samples for occupancy classification from a chessboard image. First, the original image (a) is warped to a two-dimensional overhead view, (b). Then, all squares are cropped (with a 50% increase in width and height to include contextual information). Finally, the cropped squares are annotated using the FEN groundtruth as either empty (c) or occupied (d).

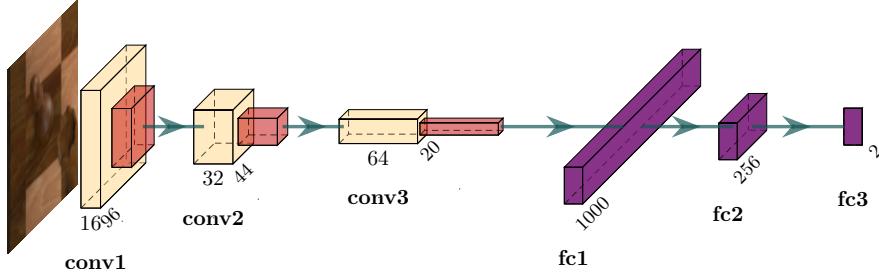


Figure 4.12: Architecture of the CNN (100, 3, 3, 3) network for occupancy classification. The input is a three-channel RGB image with  $100 \times 100$  pixels. There are two convolutional layers (yellow) with a kernel size of  $5 \times 5$  and stride 1, meaning that each convolutional layer reduces the width and height by 4. The final convolutional layer has a kernel size of  $3 \times 3$ , thus only reducing the input size by two. Starting with 16 filters in the first convolutional layer, the number of channels is doubled in each subsequent layer, as is common practice in CNNs [61]. Each convolutional layer uses the ReLU activation function and is followed by a max pooling layer with a  $2 \times 2$  kernel that is moved with a stride of 2 such that the width and height are halved. Finally, the output of the last pooling layer is reshaped to a 640,000-dimensional vector that passes through two fully connected ReLU-activated layers before reaching the final fully connected layer with softmax activation.

#### 4.2.1 Designing and training CNNs

Six CNN architectures are devised for the occupancy classification task, of which two accept  $100 \times 100$  pixel input images and the remaining four require the images to be of size  $50 \times 50$  pixels. They differ in the number of convolution layers, pooling layers, and fully connected layers. When referring to these models, we shall use a 4-tuple consisting of the input side length and the three aforementioned criteria. Figure 4.12 depicts the architecture of CNN (100, 3, 3, 3) which achieves the greatest validation accuracy of these six models. The final fully connected layer in each model contains two output units that represent the two classes (occupied and empty). The models are trained using the cross-entropy loss function on the outputs. Training proceeds using the popular *Adam* optimizer [46] with a learning rate of 0.001 for three whole passes over the training set using a batch size of 128. At every 100 steps, the model's loss and accuracy is computed over the entire validation set, the results of which are reported in fig. 4.13. The model converges smoothly to a very low loss value, achieving a training accuracy of 99.70% and validation accuracy of 99.71%. Due to the fact that the difference between training and validation accuracy is very small (in fact, the validation accuracy even happens to be slightly above the training accuracy), we conclude that the model does not overfit the training set.

#### 4.2.2 Transfer learning on deeper models

Apart from training custom CNN architectures as described in the previous section, we examine whether fine-tuning pre-trained models on our dataset can achieve even better performances. Section 2.4 explains the motivation for em-

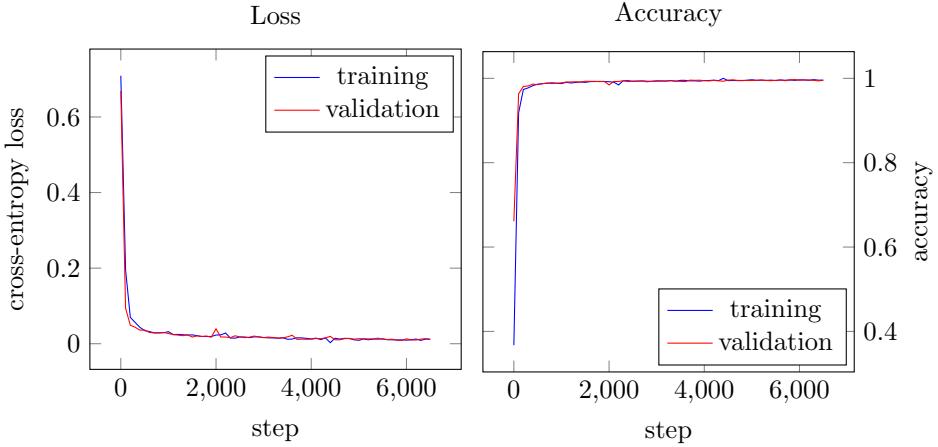


Figure 4.13: Loss and accuracy during training on both the training and validation sets for the CNN (100, 3, 3, 3) model. The best validation accuracy is 99.71%.

ploying transfer learning and the general methodology of initialising the network with the pre-trained weights, then training just the classification head and finally training the whole network. We replace the final layer of the pre-trained model’s classification head with a fully-connected layer that has two output units so the network can classify ‘empty’ and ‘occupied’ squares. Due to the abundance of data in the training set, it suffices to train the classification head for only one epoch (one pass over the dataset) with a learning rate  $\alpha = 10^{-3}$ , followed by training the whole network for two epochs<sup>1</sup> with  $\alpha = 10^{-4}$ . As underlying models, we compare the popular CNN architectures VGG [61], ResNet [62], and AlexNet [34]. The weights in each case were pretrained on the ImageNet [51] dataset.

The ResNet model achieves the highest validation accuracy (99.96%) of all evaluated architectures. Training progresses smoothly and there is no significant gap between the training and validation metrics which are depicted in fig. 4.14. The large decrease in loss and simultaneous increase in accuracy that occurs just after step 2,000 in that figure coincides with the transition from training exclusively the classification head to the whole network. After that, the model converges smoothly to a low loss and high accuracy. The final model misclassifies only four of the 146 samples in the validation set; these are shown in fig. 4.15.

### 4.2.3 Analysis

Each model is trained separately on the dataset of squares that are cropped to include contextual information (by increasing the bounding box by 50% in each direction), and the same samples except that the squares are cropped tightly. Key performance metrics for each model are summarised in table 4.1. In each case, the model trained on the samples that contained contextual information outperformed its counterpart trained on tightly cropped samples, confirming the

<sup>1</sup>The number of steps is significantly greater than the number of epochs because each epoch is split into multiple batches.

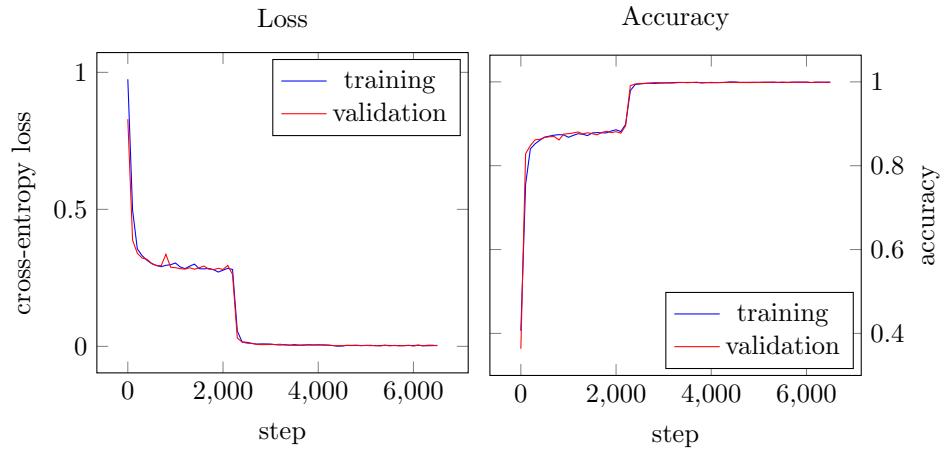


Figure 4.14: Loss and accuracy during training on both the training and validation sets for the ResNet model. The best validation accuracy is 99.96%.

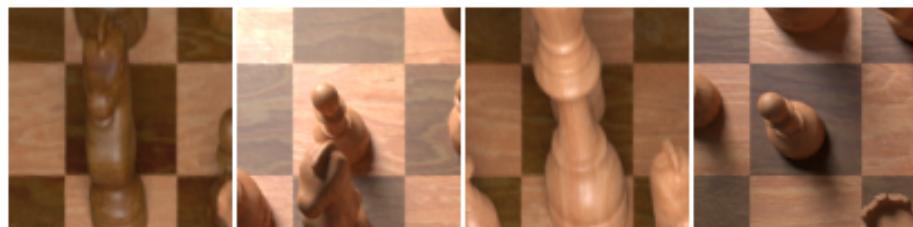


Figure 4.15: The four samples that the ResNet model misclassified in the validation set of size 146. The left sample depicts an empty square, and the remaining three are of occupied squares.

	model	trainable parameters	accuracy	precision	recall	errors	training accuracy
✓	ResNet [62]	$1.12 \cdot 10^7$	99.96%	1.000	0.999	4	99.93%
✓	VGG [61]	$1.29 \cdot 10^8$	99.95%	0.999	0.999	5	99.96%
✗	VGG [61]	$1.29 \cdot 10^8$	99.94%	0.999	0.999	6	99.93%
✗	ResNet [62]	$1.12 \cdot 10^7$	99.90%	0.999	0.998	9	99.94%
✓	AlexNet [34]	$5.7 \cdot 10^7$	99.80%	0.998	0.996	19	99.74%
✗	AlexNet [34]	$5.7 \cdot 10^7$	99.76%	0.998	0.995	22	99.76%
✓	CNN (100, 3, 3, 3)	$6.69 \cdot 10^6$	99.71%	0.997	0.995	27	99.70%
✓	CNN (100, 3, 3, 2)	$6.44 \cdot 10^6$	99.70%	0.996	0.995	28	99.70%
✗	CNN (100, 3, 3, 2)	$6.44 \cdot 10^6$	99.64%	0.996	0.993	34	99.61%
✓	CNN (50, 2, 2, 3)	$4.13 \cdot 10^6$	99.59%	0.993	0.995	38	99.62%
✓	CNN (50, 3, 1, 2)	$1.86 \cdot 10^7$	99.56%	0.997	0.991	41	99.67%
✓	CNN (50, 3, 1, 3)	$1.88 \cdot 10^7$	99.56%	0.993	0.994	41	99.66%
✓	CNN (50, 2, 2, 2)	$3.88 \cdot 10^6$	99.54%	0.993	0.994	43	99.64%
✗	CNN (50, 2, 2, 3)	$4.13 \cdot 10^6$	99.52%	0.993	0.993	45	99.57%
✗	CNN (100, 3, 3, 3)	$6.69 \cdot 10^6$	99.50%	0.988	0.997	47	99.55%
✗	CNN (50, 3, 1, 2)	$1.86 \cdot 10^7$	99.50%	0.993	0.992	47	99.44%
✗	CNN (50, 2, 2, 2)	$3.88 \cdot 10^6$	99.44%	0.995	0.989	52	99.54%
✗	CNN (50, 3, 1, 3)	$1.88 \cdot 10^7$	99.39%	0.993	0.989	57	99.41%

Table 4.1: Performance of all occupancy classification models on the validation set. For the CNN models, the 4-tuple denotes the length of the square input size in pixels, the number of convolution layers, the number of pooling layers, and the number of fully connected layers. The check mark in the left column indicates whether the input samples contained contextual information (cropped to include part of the adjacent squares). In the penultimate column, the total number of misclassifications in the validation set are reported (the validation set consists of 9,346 samples). The training accuracy is given in the rightmost column for comparison. Notice that there is no significant difference between the validation and training accuracies, indicating that none of the models suffer from overfitting.

hypothesis voiced at the beginning of section 4.2 that the information around the square itself is useful.

Furthermore, we see that the pre-trained models from section 4.2.2 perform better than the CNNs from section 4.2.1, although the differences in accuracy are small and every model achieves accuracies above 99%. Likely reasons for the superiority of the pre-trained models are the increased number of trainable parameters (up to two orders of magnitude higher than the simple CNNs), the use of transfer learning, and the more complex architectural designs. Nonetheless, we see in table 4.1 by comparing the training and validation accuracies that none of the models suffer from overfitting which is not surprising given the size of the dataset. We select the ResNet model for use in the chess recognition pipeline because it attains the highest accuracy score.

### 4.3 Piece classification

Now that the occupancy of each square on the chessboard can be detected to a high degree of accuracy, the next step is to classify the piece in each of the occupied squares. We require a 12-way classifier that takes as input a cropped image of an occupied square and will output the chess piece on that square. There are six types of chess pieces (pawn, knight, bishop, rook, queen, and king), and each piece can either be white or black in colour, thus there are a dozen classes.

We must pay some special attention to how the pieces are cropped. Simply following the approach described in section 4.2 would provide insufficient information to classify pieces. Especially tall pieces at the back of the board would be cropped in a manner such that only the bottom part of the piece remains in the region of interest (ROI). Consider for example the white king in fig. 4.11. Cropping only the square it is located in would not include its crown which is an important feature needed to distinguish between kings and queens. Instead, we must choose a rectangular bounding box that is tall enough to account for the perspective distortion.

The camera perspective causes another phenomenon that we must account for: pieces on the left of the board tend to ‘slant’ to the left, and vice-versa on the right. This is due to the vanishing point of the normal vectors on the chessboard surface being roughly centred horizontally with regards to the location of the chessboard itself<sup>2</sup>, as illustrated in fig. 4.16. Hence, we must extend the ROI horizontally in the appropriate direction.

To obtain the ROIs, we first warp the input image of the chessboard as described in section 4.2 and exemplified in fig. 4.11(b). At first, each piece’s bounding box will correspond to the square it is located on, i.e. its width and height will be that of the square<sup>3</sup>. Depending on the rank  $r$ , the height is

<sup>2</sup>Unfortunately, it is not possible to compute the vanishing point of the normals based solely on the information available in the input images as this would require knowledge of the intrinsic and extrinsic camera parameters [63]. The aim of this work is to recognise chess positions from just the input image without any further information; therefore, we devise a heuristic based on the observation that the vanishing point tends to be vertically below the chessboard and roughly horizontally centred.

<sup>3</sup>Note that due to the warped perspective, all squares have equal width and height.

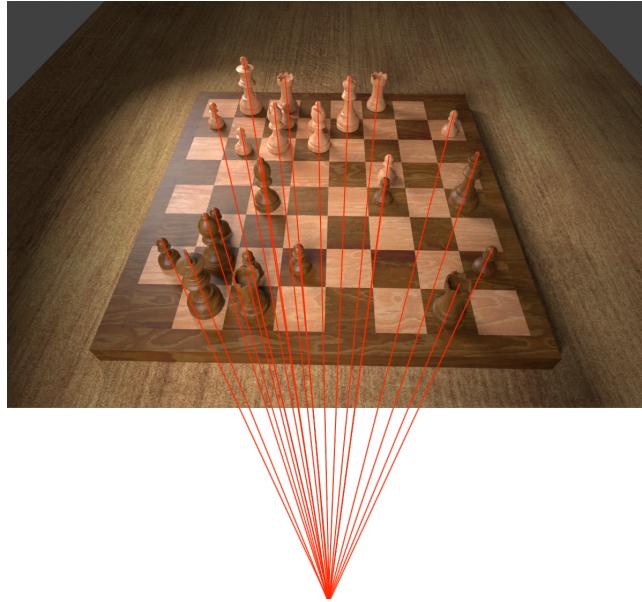


Figure 4.16: The normals of the chessboard surface (corresponding to the direction the pieces are pointing, marked in red) converge to a single vanishing point which is below the image. We will assume that the vanishing point is roughly horizontally centred with the chessboard, as this corresponds to how chessboards are usually photographed. As a result, pieces on left ‘lean’ left, and vice-versa on the right.

increased by

$$h_{\text{inc}}(r) = \frac{2r + 5}{7}$$

where the unit of measurement corresponds to the warped coordinate system described in section 4.1.2.1 (each chess square is one square unit). This represents an arithmetic progression such that the bounding box for pieces in the first rank (bottom row of the chessboard, i.e.  $r = 1$ ) will be increased by one square, and pieces in the top row ( $r = 8$ ) will have their height increased by three units.

The increase in width is dependent on the file  $f$  and given by the piecewise-defined arithmetic progression

$$w_{\text{inc}}(f) = \begin{cases} -\frac{f}{4} & f \leq 4 \\ \frac{f-4}{4} & f > 4. \end{cases}$$

A negative increase in width means that the bounding box is extended to the left, whereas a positive increase means it is widened to the right. Thus, pieces on the left half of the board ( $1 \leq f \leq 4$ ) will have their bounding boxes extended to the left, and vice-versa on the right (for  $4 < f \leq 8$ ).

Experiments demonstrate that this heuristic generates bounding boxes that are large enough to contain even the tallest pieces in most cases, whilst not being needlessly large. As a further preprocessing step, the cropped ROIs for all pieces on the left side of the board ( $1 \leq f \leq 4$ ) are flipped vertically, such that the square that the piece stands on will always be in the bottom left of the image. This may help the classifier understand what piece is being referred to

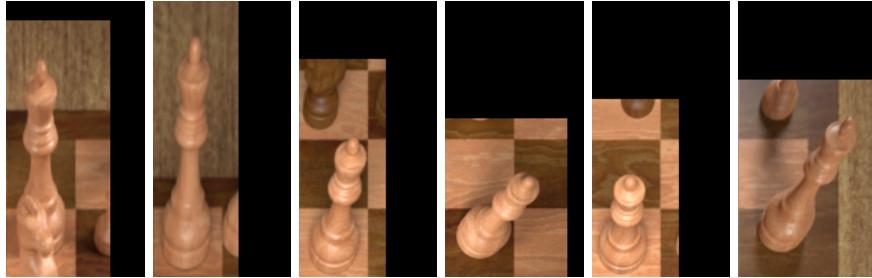


Figure 4.17: A random selection of six samples of white queens in the training set. Notice that the square each queen is located on is always in the bottom left of the image and of uniform dimensions across all samples.

model	trainable parameters	accuracy	errors	training accuracy
InceptionV3 [64]	$2.44 \cdot 10^7$	100.00%	0	99.98%
VGG [61]	$1.29 \cdot 10^8$	99.94%	2	99.84%
ResNet [62]	$1.12 \cdot 10^7$	99.91%	3	99.93%
AlexNet [34]	$5.71 \cdot 10^7$	99.02%	31	99.51%
CNN (100, 3, 3, 2)	$1.41 \cdot 10^7$	96.94%	97	99.62%
CNN (100, 3, 3, 3)	$1.44 \cdot 10^7$	96.90%	98	99.49%

Table 4.2: Performance of all piece classifiers on the validation set.

in samples where the larger bounding box includes adjacent pieces in the image. Figure 4.17 shows a random selection of the ROIs corresponding to white queens in order to demonstrate that the bounding boxes are large enough to contain even tall pieces.

### 4.3.1 CNN models

Similar to section 4.2, we train several CNNs for the piece classification task. However, since the problem is not binary anymore, the final layer of our CNNs will have 12 instead of two output units. For the pre-trained models, we follow the same two-stage training regime, but double the number of epochs at each stage, so that the classification head is trained for two epochs followed by the whole network for four epochs. Furthermore, we introduce one more architecture, InceptionV3 [64], that is a bit more complicated to train due to its use of batch normalisation and multiple losses. The exact details are, however, beyond the scope of this report. All other configurations are as described in section 4.2.

In total, we train six models, two of which are ‘vanilla’ CNNs (the best two from section 4.2 are selected for this task), and the rest are deeper models trained by means of transfer learning. The results in table 4.2 indicate a more significant difference between the hand-crafted CNNs and the deeper models (around three percentage points) than it was the case for the occupancy classifier. The InceptionV3 model achieves the best performance with a validation accuracy of 100%, i.e. there were no misclassifications in the validation set. Therefore, we will use that model for the chess recognition pipeline. Figure 4.18

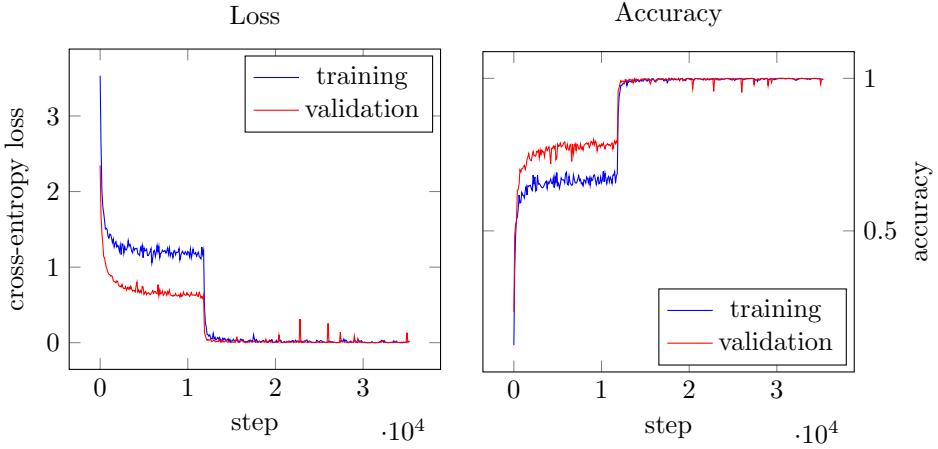


Figure 4.18: Loss and accuracy during training on both the training and validation sets for the InceptionV3 model. The best validation accuracy is 100.00%.

shows the the development of loss and accuracy during training. Similar to the ResNet occupancy classifier in fig. 4.14, we witness an abrupt drop in loss and simultaneous increase in accuracy when all remaining layers are unfrozen. This occurs earlier in the training of the piece classifier because the size of the dataset is smaller<sup>4</sup>. Figure 4.18 also exhibits a higher degree of small fluctuations around the loss and accuracy curves which can be attributed to the increased difficulty of this task where we must classify a dozen different piece types.

#### 4.4 Producing a prediction

Sections 4.1 to 4.3 describe the three components of the chess recognition pipeline. Assembling these parts facilitates an end-to-end chess recognition system that takes an input image and ultimately produces a FEN prediction. First, we localise the board (section 4.1) and obtain the pixel coordinates of the four chessboard corners. Then, we warp the input image to fit the chessboard on a regular grid. We crop each of the squares and feed them to the occupancy classifier (section 4.2). The output is a probability for the occupancy of each square. We retain only the squares with an occupancy probability over 50% and cut out the corresponding squares in the same manner as the piece classification dataset was created in section 4.3. After that, each of the occupied samples is supplied to the piece classifier in order to obtain a probability distribution over the piece types. In each instance, we simply use the class with the highest probability. Finally, we create an internal representation of the chessboard based on the results of the two classifiers in order to produce a FEN description of the position on the board.

Table 4.3 summarises key performance metrics of the end-to-end chess recognition pipeline evaluated on both the training and validation sets (the test set will be evaluated in ??). On average, the pipeline misclassifies only 0.27 squares

<sup>4</sup>The dataset for piece classification consists of only occupied squares whereas the occupancy classification dataset comprises all squares.

metric	training set	validation set
mean number of incorrect squares per board	0.27	0.27
percentage of boards predicted with no mistakes	94.77%	97.95%
percentage of boards predicted with $\leq 1$ mistake	99.14%	99.32%
per-board corner detection accuracy	99.59%	100.00%
per-square occupancy classification accuracy	99.81%	99.97%
per-square piece classification accuracy	99.99%	99.99%

Table 4.3: Performance of the chess recognition pipeline on the training and validation sets. The accuracies of the piece and occupancy classifiers are reported on a per-square basis, and in computing the accuracy of the piece classifier, we consider only the squares that were marked as ‘occupied’ by the occupancy classifier. This is because the piece classifier is used only on the occupied squares.

per board. Overall, 98% of the chessboards in the validation set are predicted without mistakes, and 2% of the samples had only one incorrect square. The accuracies of the two classifiers as well as the corner dectector is above 99%.

*The beauty of a move lies not in its appearance but in the thought behind it.*

Aron Nimzowitsch

# 5

## Adapting to new chess sets

Chess sets may vary significantly in appearance. As a result, CNNs trained on one type of chess set might perform poorly in the inference stage when supplied with images of another chess set. This is because we violate the assumption that the training and testing data are drawn from the same distribution. In light of the theory introduced in section 2.4, the natural response is to employ transfer learning in order to fine-tune the CNNs to this new data distribution. Due to the inherent similarities in the data distribution (the fact that they are chess sets and that the source and target tasks are the same), it stands to reason that we could employ a form of *one-shot* transfer learning; that is, using only a small amount of data in order to adapt the CNNs to the new distribution. Using the least amount of data necessary will make the system considerably more convenient for the user.

A significant advantage of the chess recognition system as it is developed in chapter 4 is the fact that we employ conventional computer vision techniques such as edge and line detection in order to localise the board. This means that we are able to find the location of practically *any* type of chess board with a high accuracy, not just the board used to generate the training data. On the other hand, had we employed a board localisation technique based on deep neural networks (for instance a semantic segmentation model), the system would suffer the same type of issues as the CNNs mentioned above when required to adapt to new chess sets. The fact that we can reliably find the corner points of any chess board but may not be able to correctly identify the position on the board gives rise to a system where we could employ transfer learning to obtain better predictions with only minimal extra effort on the part of the user: if the user wants to employ our system to infer a position on their own chess board, we will instruct them to take a picture of the starting position (fig. 5.1) from both players' perspectives. The angle of the camera to the chessboard should be in the same interval as used in the data synthesis process depicted in fig. 3.2, i.e. between 45° and 60°. This chapter will demonstrate how to effectively fine-tune

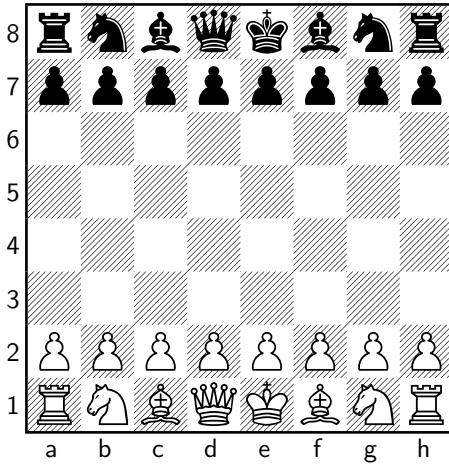


Figure 5.1: The starting position on the board. It is the same at the beginning of each game, thus a photo captured in this position is well-suited for transfer learning as the user does not need to manually label the position.



Figure 5.2: The training dataset used for the transfer learning approach, consisting of only two samples.

the occupancy and piece classification CNNs based on only two input images.

## 5.1 Dataset

First, we require a dataset in order to evaluate the effectiveness of our approach. The training set, consisting of the two images obtained in the aforementioned manner is depicted in fig. 5.2. For this task, we do not employ a validation dataset because the availability of labelled data is limited (it would be unreasonable to ask the user for more photos and then even requiring them to manually annotate the positions). The test dataset consists of 27 images which were obtained by playing a game of chess and taking a photo of the board after each move. Each photo is taken from the current player's perspective, so the images alternate between White's and Black's perspectives. All samples are associated with a manually labelled FEN string describing the position.

metric	training set	test set
mean number of incorrect squares per board	9.50	9.33
percentage of boards predicted with no mistakes	0.00%	0.00%
percentage of boards predicted with $\leq 1$ mistake	0.00%	0.00%
per-board corner detection accuracy	100.00%	100.00%
per-square occupancy classification accuracy	100.00%	99.88%
per-square piece classification accuracy	85.16%	85.52%

Table 5.1: Performance of the chess recognition pipeline from chapter 4 on the transfer learning dataset without fine-tuning. In this case, the training set is not used for training because the model is not fine-tuned.

At this point, it is a good idea to establish a baseline for the quality of predictions the chess recognition pipeline achieves without fine-tuning the CNNs on the test set. These results are summarised in table 5.1. This regime is unable to completely identify any of the positions; on average, it misclassifies 9.33 out of 64 squares per board. The accuracy of the corner detection algorithm is 100%, meaning that it can generalise nicely to new chess sets as hypothesised at the beginning of this chapter. While the occupancy classification CNN is able to adapt quite well to this new dataset as well (achieving an accuracy of 99.88% on the test set), the underlying reason for the poor performance stems from the piece classifier that achieves only around 85% accuracy.

## 5.2 Training

Our ability to fine-tune the models to this new dataset is mainly constrained by the limited availability of data. Since we only have two input images, the occupancy classifier will have 128 samples for training (each board has 64 squares). The piece classifier will only have 64 samples because there are 32 pieces on the board at the starting position. While the CNN for occupancy detection is a binary classifier, the piece classifier must undertake the more challenging task of distinguishing between a dozen different piece types. Furthermore, the data is not balanced between the classes: for example, there are 16 training samples for black pawns, but only two for the black king. Given this premise, it is clear that we should employ data augmentations at training time in order to increase the variability of the data and reduce the risk of overfitting.

### 5.2.1 Data augmentation

The use of various types of data augmentation resulted in a net increase in the accuracy of the position inference by 45 percentage points (from 44% without data augmentation to 89% with augmentation). Furthermore, the mean number errors per position was decreased from 2.3 squares to 0.11 squares. Of course, these augmentations are only applied while training the networks and not at test time. The following sections lay out various data augmentations employed to increase the performance of the piece classifier. While the occupancy classifier was fine-tuned as well, augmentations were not used as aggressively in that CNN’s training process because its accuracy was already quite high. These

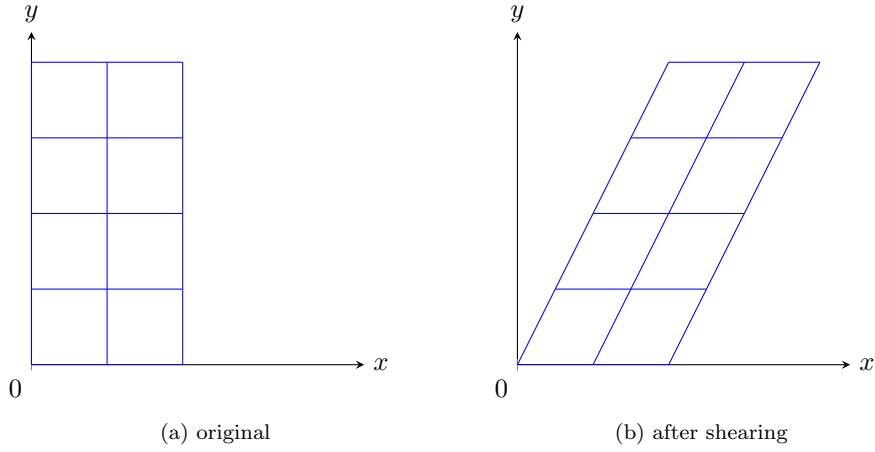


Figure 5.3: Illustration of the shear transform with  $\lambda = 1/2$ . Notice that the  $y$ -axis points in the opposite direction as is used to represent images, thus in practice, the input sample is flipped vertically before applying the shear transform and then flipped back.

augmentations had little to no effect on the performance, so in the interest of brevity we will instead focus on the augmentations of the piece classifier.

#### 5.2.1.1 Shearing

Of all augmentations, we observe the most significant performance gains in the shear transformation, likely due to its similarity to actual perspective distortion. In section 4.3, we explain that the piece images are transformed (by performing a horizontal flip where appropriate) in a manner such that the bottom left part of the sample always depicts the square on which the piece stands, as can be seen in fig. 4.17. Consequently, we must carefully design our use of the shear transform to retain this property in broad terms.

A shear transform is a linear map that disspaces pixels from the original image. In our case, the magnitude of displacement will be proportional to the  $y$ -axis, but each pixel will remain on its row. Mathematically, a shear is a function  $s : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  that maps a 2D coordinate to another. We define it as

$$s(x, y) = \begin{bmatrix} x + \lambda y \\ y \end{bmatrix} \quad (5.1)$$

for some  $\lambda \in \mathbb{R}$  affecting severity of the distortion. For  $\lambda = 0$ , there is no distortion. Figure 5.3 demonstrates the shear transform on a conceptual level and shows that the bottom left corner of the image is affected the least. At each iteration during training, we sample  $\lambda$  from a uniform distribution in the interval  $[-0.1, 0.25]$ .

#### 5.2.1.2 Affine transform

Apart from shearing, we employ two more affine transformations: translation and scaling. The random  $x$  and  $y$  displacements are sampled uniformly in the interval  $[-0.03, 0.1]$ , meaning that the piece may shift up to 3% left or down

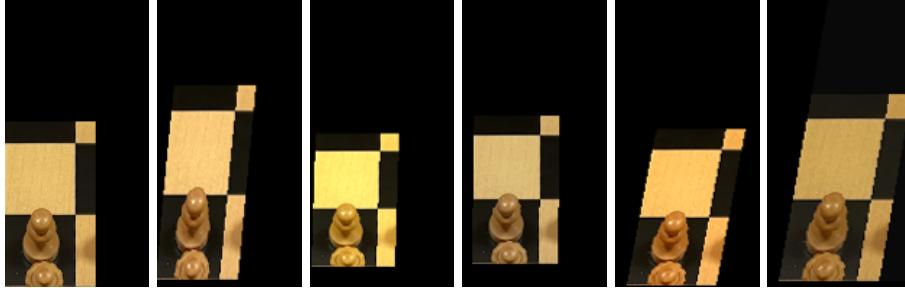


Figure 5.4: The augmentation pipeline applied to an input image (left). Each output looks different due to the random parameter selection.

(relative to the total width/height), but up to 10% up or right. Again, this is because the most important information is in the bottom left of the image.

Similar to the translation, scaling is performed separately on both axes, meaning that the  $x$  and  $y$  scales may be different. The scale factor for each axis is chosen uniformly in the interval [0.8, 1.2].

#### 5.2.1.3 Colour jitter

A popular means of image augmentation is inducing colour perturbations. We achieve this by jointly varying brightness, contrast, hue, and saturation. For each of these factors, we choose the degree of alteration from four separate uniform distributions over intervals that are sensibly selected such that the alteration does not appear too drastic upon visual inspection.

Figure 5.4 shows five augmentations of the same input image, drawing attention to the random parameter selection which creates the intended variability in the training images. The augmentation pipeline is applied to each image for each batch during training, meaning that different augmentations are used each time even though the underlying images might be the same.

#### 5.2.2 Fine-tuning

To train the networks, we follow a two-stage approach as motivated in section 2.4 and already employed in sections 4.2 and 4.3. First, we train only the classification head with a learning rate of 0.001, and then we decrease the learning rate by a factor of ten and train all layers. In the case of the occupancy classifier, we perform 100 iterations over the entire training set at both stages, essentially following the same regime as section 4.2. However, for the piece classifier, we execute an additional 50 iterations in the second stage to ensure reliable convergence. As expected, we are unable to achieve gains in the occupancy classifier’s accuracy because it already achieved such a strong performance from the outset. In fact, the loss is already below 0.001 at the beginning of training which can be seen in fig. 5.5(a). Thus the occupancy classifier achieves an accuracy of 100% quite quickly during training, even before all layers are unfrozen halfway through the training.

On the other hand, the piece classifier starts with a loss value just under 2, and only converges during the second training stage which starts around

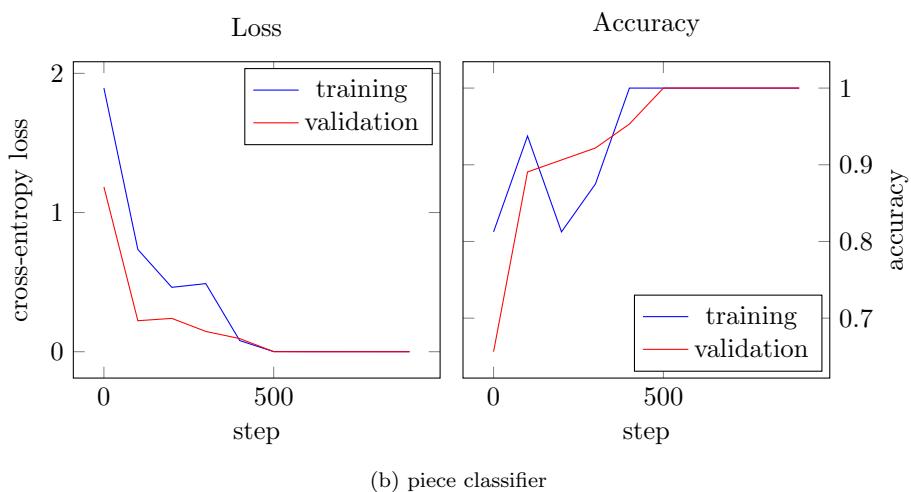
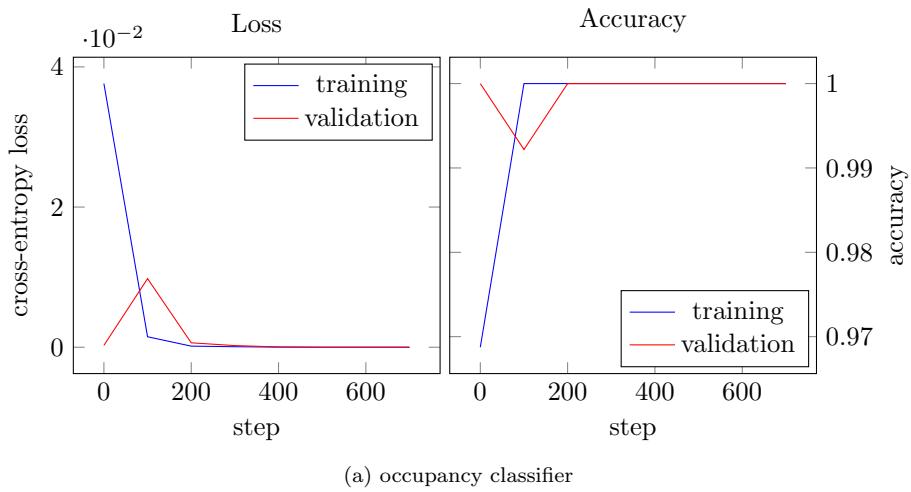


Figure 5.5: Loss and accuracy in fine-tuning the occupancy and piece classifiers on the new dataset. Due to the limited availability of data, the validation set is simply the training set without augmentations. Both fine-tuned models achieves an accuracy of 100% on the training data.

metric	training set	test set
mean number of incorrect squares per board	0.00	0.11
percentage of boards predicted with no mistakes	100.00%	88.89%
percentage of boards predicted with $\leq 1$ mistake	100.00%	100.00%
per-board corner detection accuracy	100.00%	100.00%
per-square occupancy classification accuracy	100.00%	99.88%
per-square piece classification accuracy	100.00%	99.94%

Table 5.2: Performance of the fine-tuned chess recognition pipeline on the transfer learning dataset.

step<sup>1</sup> 300, as depicted in fig. 5.5(b). Nonetheless, the convergence of the loss is relatively smooth. The fluctuations in accuracy at the beginning of training can be explained by the small size of the dataset.

Key indicators for evaluating the performance of the chess recognition pipeline using the newly fine-tuned models on the transfer learning dataset are summarised in table 5.2. Comparing that to table 5.1, we experience drastic improvements in all aspects. Even in absolute terms, the results are very strong and a testament to the effectiveness of using transfer learning and careful data augmentation in order to adapt to a slightly different data distribution. The mean number of incorrect squares per board decreased by almost two orders of magnitude to 0.11. This allowed an increase in the percentage of correctly identified positions from 0% to 89%. The remaining 11% of positions were identified with just one mistake. In other words, there were only three errors in all 27 samples in the test set.

---

<sup>1</sup>Recall that the number of steps is different to the number of iterations over the dataset, due to the fact that the dataset consists of multiple batches.

*Life is a kind of chess, in which we have often points to gain, and competitors or adversaries to contend with.*

Benjamin Franklin

# 6

## Implementation

Chapters 3 to 5 explain the design of the chess recognition system in great detail. This chapter will highlight and justify the main implementation decisions in a brief manner, as permitted by the scope of this report. For more implementation-specific details, the interested reader may consult the code directly; many important methods are described using docstrings and online documentation is available, too.

The context survey of implementation tools in section 1.1.2 justifies the use of Python as the main language of implementation. We adopt the PyTorch framework [39] to facilitate GPU-accelerated neural network training and inference due to its prevalence in the research community, and execute such tasks on a Ubuntu 20.04.1 lab machine with a 6GB NVIDIA GeForce GTX 1060 GPU. For many traditional computer vision methods employed in section 4.1 such as Canny edge detection and Hough transform, we draw on the popular OpenCV library [65] which provides efficient implementations of standard computer vision algorithms.

**Repositories** The implementation of this project took place in three main repositories, all of which were made public on GitHub<sup>1</sup>:

- **recap**<sup>2</sup>: a Python package for managing configurations and files associated with machine learning experiments.
- **chesscog**<sup>3</sup>: Python implementation of all core components of the chess recognition system: data synthesis (chapter 3), the recognition pipeline

---

<sup>1</sup>The code of each of these repositories is supplied alongside the submission, but can also be accessed at <https://github.com/georgw777/{repo}> where {repo} is to be substituted with the repository's name. The author of this report was the sole contributor to these repositories.

<sup>2</sup>*Recap* is an acronym for REproducible Configurations for Any Project, and is published as a Python package on PyPI at <http://pypi.org/project/recap>.

<sup>3</sup>The name **chesscog** is derived from its purpose: ‘chess recognition’.

including training and evaluation (chapter 4), and the transfer learning approach to adapt to new chess sets (chapter 5);

- `chesscog-app`: a proof-of-concept web app to showcase the chess inference engine; and

Originally, `recap` was part of `chesscog`; however, it was later refactored into its own repository and published on the Python Package Index (PyPI) because other people requested to use this configuration system, too. Separating the codebases of the core implementation (`chesscog`) and the web app (`chesscog-app`) ensures a clear separation of concerns and facilitates easier testing of both components in isolation.

The quality of a Python package is quite obviously predicated in part on the quality of its documentation. Therefore, classes and methods in the `recap` package and `chesscog` project are documented using not only type annotations in line with the Python Enhancement Proposal (PEP)<sup>4</sup> 484 and 526 standards, but also docstrings according to PEP 257. This enables the automatic generation of comprehensive documentation which is supplied alongside the project submission as explained in appendices B.1 and C.1.

**Continuous integration and delivery** All three repositories contain unit tests that are executed on every commit via continuous integration (CI) pipelines implemented with *Github Actions*. Running automated builds and tests that frequently enables bugs and incompatibilities to be identified quickly. This is especially useful due to the fact that the chess recognition system was developed in parallel with the web app. Figure 6.1 provides an overview of the automated pipelines and their interaction between the repositories. Whenever a release is published in the `chesscog` repository, it notifies the `chesscog-app` repository which opens a pull request (PR) where the version of the `chesscog` dependency is updated. As with a commit, this causes an automatic build and execution of the tests. If the developer is satisfied with the PR, it can be merged with one click on the GitHub web interface.

The `chesscog-app` repository has a continuous delivery (CD) pipeline that on every push to the `master` branch (provided that it passes the build and tests) will build a self-contained Docker image of the app and push that to the web server, thereby updating the hosted web app. Furthermore, every release in the `recap` repository activates a CD pipeline that publishes the new version of the package on PyPI. Both CD pipelines are depicted in fig. 6.1 as well.

The pipelines perform so-called *matrix builds* which means that the build and test stages are executed for all supported Python versions in parallel<sup>5</sup>. This is especially important for the `recap` package which is listed on PyPI as supporting Python version  $\geq 3.6$ , so the matrix builds target versions 3.6, 3.7, and 3.8.

---

<sup>4</sup>Python Enhancement Proposals are design documents containing best practices for developing Python code.

<sup>5</sup>In the case of the web app which uses Node.js in addition to Python, the matrix builds apply to the supported Node.js versions as well.

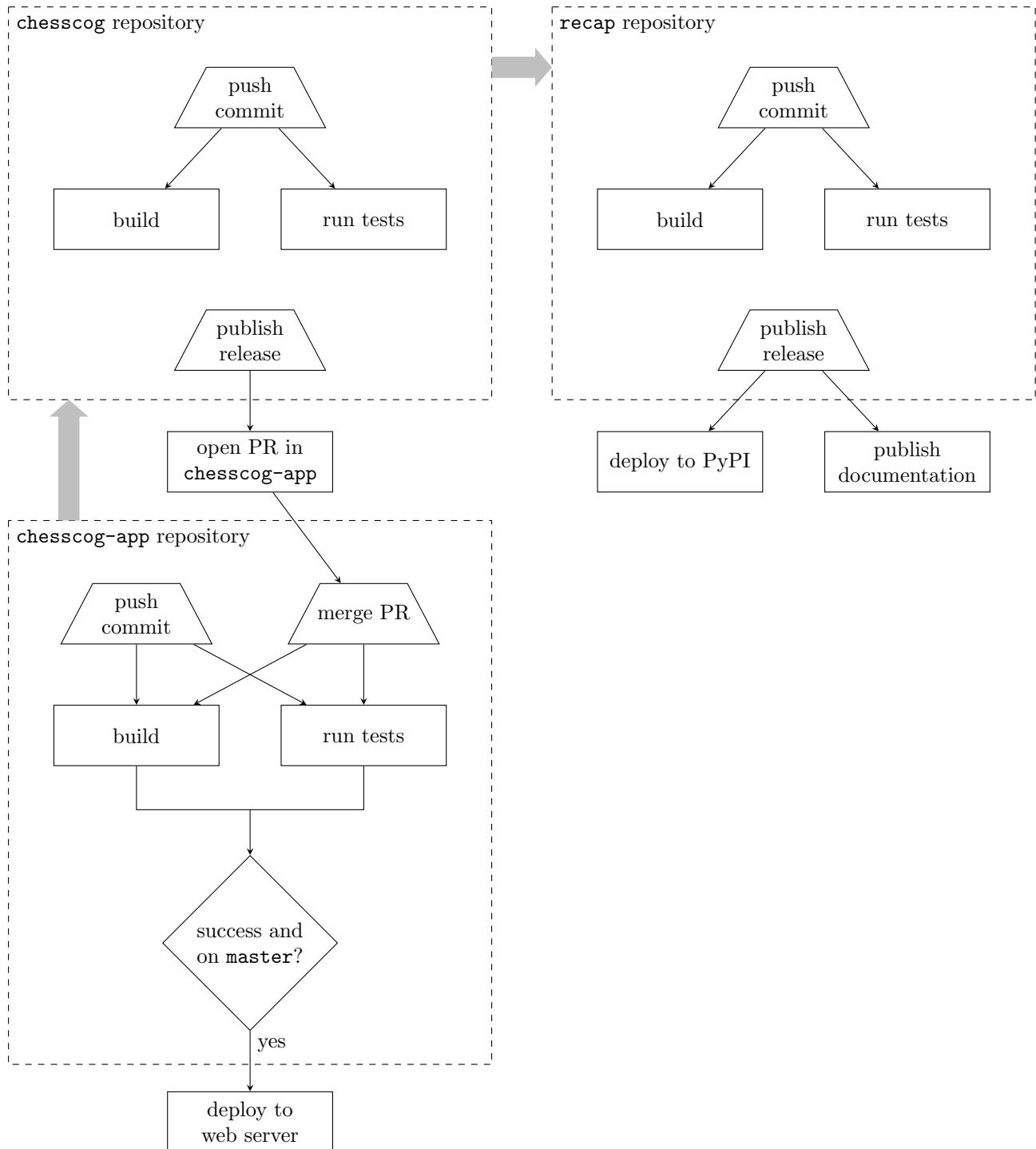


Figure 6.1: Overview of CI/CD pipelines. Trapeziums represent actions initiated by the developer, rectangles correspond to automated processes executed by the pipeline, and the thick arrows represent logical dependency. The current status of each pipeline is available at <https://github.com/georgw777/{repo}/actions> where {repo} is to be substituted with the repository's name.

## 6.1 Configuration management (recap)

A core principle underlying the implementation of the chess recognition system is that of reproducibility. All experimental results and metrics that are claimed in this report should be independently verifiable. To this end, a Python package was developed that serves the purposes of (i) persisting hyperparameters and other configuration settings of any experiment, algorithm, or system that is configurable; (ii) providing a convenient interface for accessing these configurations; and (iii) facilitate a unified means of specifying and managing paths throughout the project. This package is made available on PyPI because ensuring the reproducibility of experiments is a common issue in machine learning research.

The configurations themselves are specified in YAML format and a core feature of `recap` is the ability to inherit from other configuration files to override settings. A common use for such a configuration file is to specify how to train a CNN including the architecture and hyperparameters such as the learning rate, number of epochs, etc. Within `chesscog`, configuration files are mainly used to specify the parameters of the corner detection system (including its sub-algorithms such as Canny edge detection and Hough transform) and the training of the two CNN classifiers. Furthermore, YAML files were automatically generated in the grid search explained in section 4.1.4. The functionality of inheriting from parent configurations was useful in section 5.2.2 where some hyperparameters (the number of epochs) and settings (the path to the dataset) were altered for training the CNNs for the new dataset, but the remaining configuration stayed the same. Finally, the weights of the exported CNN models are supplied alongside the state of the configuration at training time to ensure the same settings (apart from data augmentations) are used for inference.

Another core feature of the configuration system is a custom way of specifying paths. Instead of using relative or absolute paths<sup>6</sup> in the configuration files, `recap` utilises a format similar<sup>7</sup> to URIs (although the use of conventional paths is still possible). First, the user registers so-called *path handlers* that translate specific `recap` URIs to absolute or relative paths. Path handlers are registered for a specific URI scheme<sup>8</sup>, so if we create a path handler for the `data://` scheme pointing to `/path/to/dataset`, the `recap` URI `data://images/train` will be equivalent to `/path/to/dataset/images/train`. These URIs can be used in the configuration files as well as the code itself; `recap` provides the `URI` class which is fully compatible with Python's native `pathlib.Path` interface and lazily translates the URIs to paths on the host system.

A comprehensive documentation of the `recap` library is available at <https://recap.readthedocs.io>. This documentation is automatically published via a CD pipeline as indicated in fig. 6.1.

<sup>6</sup>Both variants are associated with issues: absolute paths may differ between machines, and relative paths break when certain dependencies (such as the dataset) are located somewhere else.

<sup>7</sup>The format of `recap` paths does not completely follow the requirements of a uniform resource locator (URI), but we will still refer to them as ‘`recap` URIs’.

<sup>8</sup>The scheme of a URI is the part before the colon and double forward slash (`://`).

## 6.2 Chess recognition system (`chesscog`)

The chess recognition system was implemented incrementally in small parts. Development usually started off in an interactive Python notebook for rapid prototyping and once a working sub-algorithm was implemented, the notebook was refactored to one or more Python files. Keeping the code in Python files instead of notebooks is a prerequisite for the automated unit tests and is furthermore necessary to avoid code duplication because many parts of the chess recognition system are used in both the training and inference stages. This also allowed the code to be structured in the form of a Python package so it can be installed directly from the current version of the repository’s `master` branch via `pip`<sup>9</sup>. Many of the Python files simultaneously act as scripts with a simple command line interface (CLI). Their usage is explained in the user manual in appendix C. The `chesscog` repository also provides a `Dockerfile` (alongside a Docker Compose configuration) to build a container to perform GPU-accelerated neural network training on the dedicated lab machine and inspect progress using the Tensorboard tool.

The 3D chess set used to synthesise the dataset was modelled using the Blender software [66]. Blender provides a Python interface which was used to automate the placement of the chess pieces on the board as well as the lighting and camera as described in see section 3.2. Apart from rendering the scenes, that same Python script was responsible for generating the associated labels as outlined in section 3.3. Appendix C.2 provides instructions for running this script. The synthesised dataset itself is provided online for download<sup>10</sup>.

In the spirit of reproducibility, we provide the trained models’ weights and relevant configuration files (created using `recap`) for download as well<sup>11</sup>. For instructions, refer to appendix C.

## 6.3 Web app (`chesscog-app`)

The web app consists of two parts: (i) the frontend user interface (UI), written in TypeScript using the popular React framework; and (ii) a simple representational state transfer (REST) application programming interface (API) for performing the chess position inference via the `chesscog` package. React apps are typically run using Node, but since all the code client-side, we bundle the app to a set of static HTML files that are served using `nginx`. The backend API was developed using the `fastapi` Python library which implements the asynchronous server gateway interface (ASGI). It provides one main route<sup>12</sup>, `/api/predict`, which upon receiving a HTTP POST request with an input image, will run the chess recognition pipeline and return the predicted FEN position alongside some other information in JSON format. Figure 6.2 provides an overview of the web app’s main components and their communication. The

---

<sup>9</sup>`pip` is the Python package installer.

<sup>10</sup>The dataset ( $\approx 7\text{GB}$ ) is available for download at <https://tinyurl.com/chesscog-dataset> or by executing the Python script for downloading it as instructed in appendix C.2.

<sup>11</sup>The files are quite large, so they are available for download rather than supplying them in the project submission or Git repository.

<sup>12</sup>There is an interactive API documentation available at <https://www.chesscog.com/api/docs>.

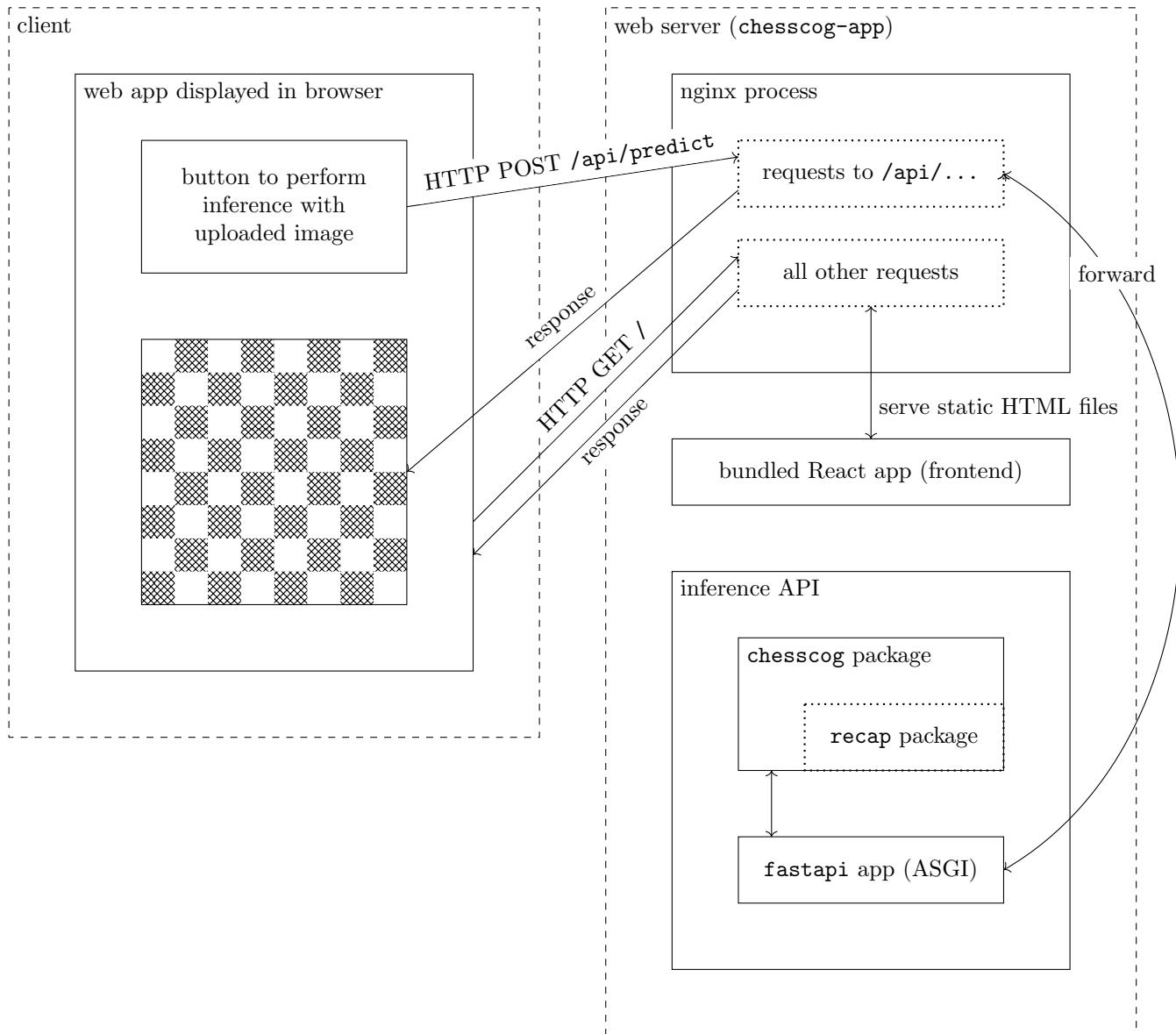


Figure 6.2: Schematic overview of the infrastructure pertaining to the web app.

architecure is quite basic and is not optimised for scale because its purpose is merely to provide a proof of concept.

The web app is hosted with a free plan on the Heroku cloud platform. As such, the server has no GPU and thus the inference of CNNs takes much longer than on the lab machine (chapter 7 provides a benchmark for inference speeds on a GPU). A CD pipeline deploys the app by building a Docker image of the system described in fig. 6.2 and pushing it to Heroku which makes it available at <https://chesscog.herokuapp.com>. The domain name system (DNS) provider for the [www.chesscog.com](http://www.chesscog.com) domain has a canonical name (CNAME) record that points to [chesscog.herokuapp.com](https://chesscog.herokuapp.com)<sup>13</sup>, so the website is accessible at <https://www.chesscog.com>. A transport layer security (TLS) certificate is installed for the web app to ensure that it supports HTTPS in addition to HTTP. Appendix D provides more information about the web app.

---

<sup>13</sup>Technically, the CNAME record points at a Heroku DNS server that forwards to [chesscog.herokuapp.com](https://chesscog.herokuapp.com).

*Chess is the struggle against the error.*

Johannes Zukertort

# 7

## Evaluation

In chapter 4, we evaluated the performance of each component in the chess recognition pipeline separately as well as the system as a whole, but that analysis was limited to the training and validation sets. The test set remained untouched thus far for reasons concerning data leakage outlined in section 3.4. Now we can evaluate the chess recognition system on the held-out test set because we have ensured that the models never before encountered this data. Table 7.1 lists key evaluation metrics for the test set and contains the corresponding numbers from the other two datasets (as in table 4.3) for comparison. There is no indication of overfitting because there are only slight differences in the results of the train and test sets. The end-to-end per-board accuracy of our system on the unseen test set is 93.86%, and if we allow just one mistake on the board, that accuracy increases to 99.42%. Comparing the overall accuracy to the training set, we see a decrease in almost one percentage point. It appears that the reason for the slightly worse performance on the test set is the marginally lower accuracy of the board localisation algorithm (99.59% on the training set vs. 99.42% on the test set). The two CNNs actually perform on par or even better on the test set

metric	train	val	test
mean number of incorrect squares per board	0.27	0.27	0.24
percentage of boards predicted with no mistakes	94.77%	97.95%	93.86%
percentage of boards predicted with $\leq 1$ mistake	99.14%	99.32%	99.42%
per-board corner detection accuracy	99.59%	100.00%	99.42%
per-square occupancy classification accuracy	99.81%	99.97%	99.92%
per-square piece classification accuracy	99.99%	99.99%	99.99%

Table 7.1: Performance of the chess recognition system on the test dataset. The training and validation metrics as per table 4.3 are included for comparison.

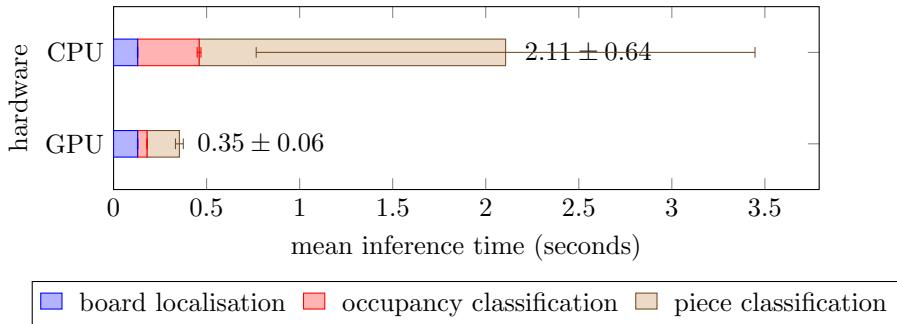


Figure 7.1: Inference time benchmarks of the chess recognition pipeline on the test set, averaged per sample. The error bars indicate the standard deviation and the mean total inference time is on the right of the bars. All benchmarks were carried out on the same machine (the Ubuntu lab machine described in ??), although the data for the trial labelled CPU was gathered without GPU acceleration.

than the training set.

The empirical results on the test set clearly demonstrate that the chess recognition system is highly accurate. However, for such a system to be practically effective, it must also be fast enough to perform an inference in a short amount of time. To this end, we record the inference time for each of the test set samples and compute the mean over all samples. In fact, we can provide more granular insights by additionally recording the average time in executing each of the three stages in the pipeline. We conduct this experiment twice on the dedicated lab machine: in the first trial, we use only the central processing unit (CPU) whereas we enable GPU acceleration for the second run. The machine is equipped with a quad-core 3.20GHz Intel Core i5-6500 CPU, and, as mentioned in ??, a 6GB NVIDIA GeForce GTX 1060 GPU. Figure 7.1 shows that the system is around six times faster when allowed to use the GPU. This is to be expected because the forward pass through the neural network executes many matrix operations that follow the SIMD amenable

## 7.1 Achievement of objectives

## 7.2 Critical appraisal

*You may learn much more from a game you lose than from a game you win.*

José Capablanca

# 8

## Conclusion

chess ever more popular covid pandemic

### 8.1 Future work

# Acronyms

<b>ANN</b> artificial neural network	5, 7, 9–11, 15, 16	<b>MLP</b> multi-layer perceptron	v, 12, 13, 15, 16, 18, 19, 21
<b>API</b> application programming interface	65, 66	<b>MSE</b> mean squared error	13
<b>ASGI</b> asynchronous server gateway interface	65, 66	<b>OHE</b> one-hot encoding	8
<b>CAD</b> computer-aided design	4	<b>PEP</b> Python Enhancement Proposal	62
<b>CD</b> continuous delivery	vi, 62–64, 67	<b>PMF</b> probability mass function	8
<b>CI</b> continuous integration	vi, 62, 63	<b>PR</b> pull request	62, 63
<b>CLI</b> command line interface	65	<b>PyPI</b> the Python Package Index	61–64
<b>CNAME</b> canonical name	67	<b>RANSAC</b> random sample consensus	34–36
<b>CNN</b> convolutional neural network	3–5, 7, 16, 17, 19, 20, 29, 45, 46, 48, 49, 51, 54–56, 64, 67, 68	<b>ReLU</b> rectified linear unit	10, 45
<b>CPU</b> central processing unit	69	<b>REST</b> representational state transfer	65
<b>DAG</b> directed acyclic graph	10, 12	<b>RGB</b> red, green, blue	15, 29, 30, 39, 45
<b>DNS</b> domain name system	67	<b>ROI</b> region of interest	49–51
<b>FEN</b> Forsyth–Edwards Notation [3]	6, 22, 23, 26, 29, 44, 52, 55, 65	<b>SIFT</b> scale-invariant feature transform	3
<b>GPU</b> graphics processing unit	5, 20, 61, 65, 67, 69	<b>SIMD</b> single instruction, multiple data	5, 69
<b>HOG</b> histogram of oriented gradients	3	<b>SLP</b> single-layer perceptron	v, 11, 12, 14, 15
<b>HTML</b> hypertext markup language	65, 66	<b>SVM</b> support vector machine	2, 3
<b>HTTP</b> hypertext transfer protocol	65–67	<b>TLS</b> transport layer security	67
<b>HTTPS</b> hypertext transfer protocol secure	67	<b>UI</b> user interface	65
<b>JSON</b> JavaScript object notation	26, 28, 65	<b>URI</b> uniform resource locator	64
		<b>YAML</b> YAML ain't markup language	64, 71

# Bibliography

- [1] I. Goodfellow, Y. Bengio and A. Courville, *Deep Learning*. MIT Press, 2016.
- [2] G. Kasparov and M. Greengard, *Deep Thinking: Where Machine Intelligence Ends and Human Creativity Begins*. 2018, ISBN: 978-1-4736-5351-1.
- [3] S. J. Edwards, *PGN Standard*. Mar. 1994. [Online]. Available: <http://archive.org/details/pgn-standard-1994-03-12>.
- [4] H. Baird and K. Thompson, ‘Reading chess,’ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 6, pp. 552–559, Jun. 1990.
- [5] I. M. Khater, A. S. Ghorab and I. A. Aljarrah, ‘Chessboard recognition system using signature, principal component analysis and color information,’ in *International Conference on Digital Information Processing and Communications*, Jul. 2012, pp. 141–145.
- [6] A. Sameer, *Tensorflow\_chessbot*, Sep. 2020. [Online]. Available: [https://github.com/Elucidation/tensorflow\\_chessbot](https://github.com/Elucidation/tensorflow_chessbot).
- [7] A. Roy, *Chessputzer*, Jul. 2020. [Online]. Available: <https://github.com/metterklume/chessputzer>.
- [8] V. Wang and R. Green, ‘Chess move tracking using overhead RGB web-cam,’ in *International Conference on Image and Vision Computing New Zealand*, Nov. 2013, pp. 299–304.
- [9] T. Cour, R. Lauranson and M. Vachette, ‘Autonomous Chess-playing Robot,’ École Polytechnique, Palaiseau, France, Jul. 2002. [Online]. Available: <http://www.timotheecour.com/papers/ChessAutonomousRobot.pdf>.
- [10] D. Urting and Y. Berbers, ‘MarineBlue: A low-cost chess robot,’ in *International Conference Robotics and Applications*, Salzburg, Austria, Jun. 2003, pp. 76–81.
- [11] N. Banerjee, D. Saha, A. Singh and G. Sanyal, ‘A Simple Autonomous Chess Playing Robot for playing Chess against any opponent in Real Time,’ in *International Conference on Computational Vision and Robotics*, vol. 58, Bhubaneshwar, India: Interscience Research Network, Aug. 2012, pp. 17–22.
- [12] A. T.-Y. Chen and K. I.-K. Wang, ‘Computer vision based chess playing capabilities for the Baxter humanoid robot,’ in *International Conference on Control, Automation and Robotics*, Apr. 2016, pp. 11–14.

## BIBLIOGRAPHY

---

- [13] J. Gonçalves, J. Lima and P. Leitão, ‘Chess robot system : A multi-disciplinary experience in automation,’ in *Spanish Portuguese Congress on Electrical Engineering*, 2005.
- [14] R. A. M. Khan and R. Kesavan, ‘Design and development of autonomous chess playing robot,’ *International Journal of Innovative Science, Engineering & Technology*, vol. 1, no. 1, 2014.
- [15] A. T.-Y. Chen and K. I.-K. Wang, ‘Robust Computer Vision Chess Analysis and Interaction with a Humanoid Robot,’ *Computers*, vol. 8, no. 1, p. 14, 1 Mar. 2019.
- [16] C. Matuszek, B. Mayton, R. Aimi, M. P. Deisenroth, L. Bo, R. Chu, M. Kung, L. LeGrand, J. R. Smith and D. Fox, ‘Gambit: An autonomous chess-playing robotic system,’ in *IEEE International Conference on Robotics and Automation*, May 2011, pp. 4291–4297.
- [17] E. Sokic and M. Ahic-Djokic, ‘Simple Computer Vision System for Chess Playing Robot Manipulator as a Project-based Learning Example,’ in *IEEE International Symposium on Signal Processing and Information Technology*, Dec. 2008, pp. 75–79.
- [18] J. Hack and P. Ramakrishnan, ‘CVChess: Computer Vision Chess Analytics,’ Stanford University, 2014. [Online]. Available: [https://web.stanford.edu/class/cs231a/prev\\_projects\\_2015/chess.pdf](https://web.stanford.edu/class/cs231a/prev_projects_2015/chess.pdf).
- [19] J. Ding. (2016). ‘ChessVision : Chess Board and Piece Recognition,’ [Online]. Available: [https://web.stanford.edu/class/cs231a/prev\\_projects\\_2016/CS\\_231A\\_Final\\_Report.pdf](https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf).
- [20] C. Danner and M. Kafafy. (2015). ‘Visual Chess Recognition,’ [Online]. Available: [https://web.stanford.edu/class/ee368/Project\\_Spring-1415/Reports/Danner\\_Kafafy.pdf](https://web.stanford.edu/class/ee368/Project_Spring-1415/Reports/Danner_Kafafy.pdf).
- [21] Y. Xie, G. Tang and W. Hoff, ‘Chess Piece Recognition Using Oriented Chamfer Matching with a Comparison to CNN,’ in *IEEE Winter Conference on Applications of Computer Vision*, Mar. 2018, pp. 2001–2009.
- [22] A. Halevy, P. Norvig and F. Pereira, ‘The unreasonable effectiveness of data,’ *IEEE Intelligent Systems*, vol. 24, pp. 8–12, 2009.
- [23] A. De la Escalera and J. M. Armingol, ‘Automatic Chessboard Detection for Intrinsic and Extrinsic Camera Parameter Calibration,’ *Sensors*, vol. 10, no. 3, pp. 2027–2044, 3 Mar. 2010.
- [24] S. Bennett and J. Lasenby, ‘ChESS – Quick and robust detection of chessboard features,’ *Computer Vision and Image Understanding*, vol. 118, pp. 197–210, Jan. 2014.
- [25] K. Tam, J. Lay and D. Levy, ‘Automatic Grid Segmentation of Populated Chessboard Taken at a Lower Angle View,’ in *Digital Image Computing: Techniques and Applications*, Dec. 2008, pp. 294–299.
- [26] J. E. Neufeld and T. S. Hall, ‘Probabilistic location of a populated chessboard using computer vision,’ in *IEEE International Midwest Symposium on Circuits and Systems*, Aug. 2010, pp. 616–619.
- [27] R. Kanchibail, S. Suryaprakash and S. Jagadish, ‘Chess Board Recognition,’ 2016. [Online]. Available: <http://vision.soic.indiana.edu/b657/sp2016/projects/rkanchib/paper.pdf>.

## BIBLIOGRAPHY

---

- [28] Y. Xie, G. Tang and W. Hoff, ‘Geometry-based populated chessboard recognition,’ in *International Conference on Machine Vision*, vol. 10696, International Society for Optics and Photonics, Apr. 2018, p. 1 069 603.
- [29] Y.-A. Wei, T.-W. Huang, H.-T. Chen and J. Liu, ‘Chess recognition from a single depth image,’ in *IEEE International Conference on Multimedia and Expo*, Jul. 2017, pp. 931–936.
- [30] M. A. Czyzewski, A. Laskowski and S. Wasik. (Jun. 2020). ‘Chessboard and chess piece recognition with the support of neural networks.’ arXiv: 1708.03898.
- [31] T. Romstad, M. Costalba and J. Kiiski, *Stockfish*, Sep. 2020. [Online]. Available: <https://github.com/official-stockfish/Stockfish>.
- [32] M. Acher and F. Esnault. (Apr. 2016). ‘Large-scale Analysis of Chess Games with Chess Engines: A Preliminary Report.’ arXiv: 1607.04186.
- [33] A. Mehta and H. Mehta, ‘Augmented Reality Chess Analyzer (ARChess-Analyzer): In-Device Inference of Physical Chess Game Positions through Board Segmentation and Piece Recognition using Convolutional Neural Networks,’ *Journal of Emerging Investigators*, Jul. 2020.
- [34] A. Krizhevsky, I. Sutskever and G. E. Hinton, ‘ImageNet classification with deep convolutional neural networks,’ *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May 2017.
- [35] M. A. Czyzewski, A. Laskowski and S. Wasik, ‘LATCHESS21: Dataset of damaged chessboard lattice points (chessboard features) used to train LAPS detector (grayscale/21x21px),’ RepOD, 2018.
- [36] J. Hou, ‘Chessman Position Recognition Using Artificial Neural Networks.’
- [37] T. Elliott. (Jan. 2019). ‘The state of the octoverse,’ The GitHub Blog, [Online]. Available: <https://github.blog/2019-01-24-the-state-of-the-octoverse-machine-learning/>.
- [38] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu and X. Zheng, ‘TensorFlow: Large-scale machine learning on heterogeneous systems,’ 2015.
- [39] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala, ‘PyTorch: An imperative style, high-performance deep learning library,’ in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.
- [40] H. He. (Oct. 2019). ‘The State of Machine Learning Frameworks in 2019,’ The Gradient, [Online]. Available: <https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/>.

## BIBLIOGRAPHY

---

- [41] A. Burkov, *The Hundred-Page Machine Learning Book*. Quebec, Canada: Andriy Burkov, 2019.
- [42] W. S. McCulloch and W. Pitts, ‘A logical calculus of the ideas immanent in nervous activity,’ *Bulletin of Mathematical Biophysics*, Dec. 1943.
- [43] X. Glorot, A. Bordes and Y. Bengio, ‘Deep Sparse Rectifier Neural Networks,’ in *International Conference on Artificial Intelligence and Statistics*, JMLR Workshop and Conference Proceedings, Jun. 2011.
- [44] S. J. Russell, P. Norvig and E. Davis, *Artificial Intelligence: A Modern Approach*, 3rd. Upper Saddle River: Prentice Hall, 2010.
- [45] Y. LeCun, Y. Bengio and G. Hinton, ‘Deep learning,’ *Nature*, vol. 521, no. 7553, pp. 436–444, 7553 May 2015.
- [46] D. P. Kingma and J. Ba. (Jan. 2017). ‘Adam: A Method for Stochastic Optimization.’ arXiv: 1412.6980.
- [47] K. Fukushima, ‘Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,’ *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, Apr. 1980.
- [48] Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, ‘Gradient-based learning applied to document recognition,’ *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
- [49] D. H. Hubel and T. N. Wiesel, ‘Receptive fields of single neurones in the cat’s striate cortex,’ *The Journal of Physiology*, vol. 148, no. 3, pp. 574–591, 1959.
- [50] S. Kornblith, M. Norouzi, H. Lee and G. Hinton, ‘Similarity of neural network representations revisited,’ in *International Conference on Machine Learning (ICML)*, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97, Long Beach, California, USA: PMLR, Jun. 2019, pp. 3519–3529.
- [51] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li and L. Fei-Fei, ‘ImageNet: A large-scale hierarchical image database,’ in *IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255.
- [52] M. Bilalić, R. Langner, M. Erb and W. Grodd, ‘Mechanisms and neural basis of object and pattern recognition: A study with chess experts,’ *Journal of Experimental Psychology*, vol. 139, no. 4, pp. 728–742, 2010.
- [53] Q. Zhou. (May 2018). ‘Pattern recognition in chess,’ ChessBase, [Online]. Available: <https://en.chessbase.com/post/pattern-recognition-in-chess>.
- [54] 64 Squares. (Feb. 2020). ‘Magnus Carlsen Chess Games,’ PGN Mentor, [Online]. Available: <https://www.pgnmentor.com/players/Carlsen/>.
- [55] J. Canny, ‘A Computational Approach to Edge Detection,’ *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986.
- [56] P. V. C. Hough, ‘Method and means for recognizing complex patterns,’ U.S. Patent 3069654A, Dec. 1962.
- [57] R. O. Duda and P. E. Hart, ‘Use of the Hough transformation to detect lines and curves in pictures,’ *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, Jan. 1972.

## BIBLIOGRAPHY

---

- [58] M. Ester, H.-P. Kriegel, J. Sander and X. Xu, ‘A density-based algorithm for discovering clusters in large spatial databases with noise,’ in *International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, Aug. 1996.
- [59] R. Szeliski, ‘Image formation,’ in *Computer Vision: Algorithms and Applications*, London: Springer, 2011, pp. 27–86, ISBN: 978-1-84882-935-0.
- [60] M. A. Fischler and R. C. Bolles, ‘Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography,’ *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, Jun. 1981.
- [61] K. Simonyan and A. Zisserman, ‘Very Deep Convolutional Networks for Large-Scale Image Recognition,’ in *International Conference on Learning Representations*, San Diego, USA, May 2015.
- [62] K. He, X. Zhang, S. Ren and J. Sun, ‘Deep Residual Learning for Image Recognition,’ in *IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2016, pp. 770–778.
- [63] R. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, 2nd ed. Cambridge: Cambridge University Press, 2004.
- [64] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, ‘Rethinking the Inception Architecture for Computer Vision,’ in *IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2016, pp. 2818–2826.
- [65] G. Bradski, ‘The OpenCV library,’ *Dr. Dobb’s Journal of Software Tools*, 2000.
- [66] Blender Online Community, *Blender - a 3D modelling and rendering package*, manual, Blender Foundation, Stichting Blender Foundation, Amsterdam, 2020. [Online]. Available: <http://www.blender.org>.

# A

## Testing summary

# B

## User manual: recap package

---

### B.1 Documentation

---

# C

## User manual: chesscog

### C.1 Documentation

### C.2 Data synthesis

```
cd /Applications/Blender.app/Contents/Resources/2.90/python/bin  
./python3.7m -m ensurepip  
./python3.7m -m pip install --upgrade pip  
./python3.7m -m pip install python-chess
```

download  
dataset

# D

User manual: web app



## **Ethics self-assessment form**

There are no ethical issues raised by this project. The self-assessment form is attached on the next page.

UNIVERSITY OF ST ANDREWS  
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)  
SCHOOL OF COMPUTER SCIENCE  
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.

Tick one box

- Staff Project**  
 **Postgraduate Project**  
 **Undergraduate Project**

Title of project

Identifying chess positions using machine learning

Name of researcher(s)

Georg Wölflein

Name of supervisor (for student research)

Dr Oggie Arandjelović

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted YES  NO

There are no ethical issues raised by this project

Signature Student or Researcher



Print Name

Georg Wölflein

Date

11.09.2020

Signature Lead Researcher or Supervisor



Print Name

Ognjen Arandjelovic

Date

15/09/2020

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

## Computer Science Preliminary Ethics Self-Assessment Form

### Research with human subjects

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

**YES**  **NO**

If YES, full ethics review required

For example:

Will you be surveying, observing or interviewing human subjects?

Will you be analysing secondary data that could significantly affect human subjects?

Does your research have the potential to have a significant negative effect on people in the study area?

### Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

**YES**  **NO**

If YES, full ethics review required

For example:

Is there any potential that there could be physical harm for anyone involved in the research?

Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

### Conflicts of interest

Do any conflicts of interest arise?

**YES**  **NO**

If YES, full ethics review required

For example:

Might research objectivity be compromised by sponsorship?

Might any issues of intellectual property or roles in research be raised?

### Funding

Is your research funded externally?

**YES**  **NO**

If YES, does the funder appear on the ‘currently automatically approved’ list on the UTREC website?

**YES**  **NO**

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

### Research with animals

Does your research involve the use of living animals?

**YES**  **NO**

If YES, your proposal must be referred to the University’s Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages

<http://www.st-andrews.ac.uk/utrec/>