School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Final Report

Chessboard State Prediction with Multitask Learning

Calum Wallbridge

Submitted in accordance with the requirements for the degree of Computer Science

2021/22

<Module code and name>

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	Uploaded to Minerva (DD/MM/YY)
<Example> Scanned participant consent forms	PDF file / file archive	Uploaded to Minerva (DD/MM/YY)
<Example> Link to online code repository	URL	Sent to supervisor and assessor (DD/MM/YY)
<Example> User manuals	PDF file	Sent to client and supervisor (DD/MM/YY)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) ____________________

Summary

<Concise statement of the problem you intended to solve and main achievements (no more than one A4 page)>

Immediatley explain what you've added compared to other implementations. beats the strongest openly available models.

Acknowledgements

<The page should contain any acknowledgements to those who have assisted with your work. Where you have worked as part of a team, you should, where appropriate, reference to any contribution made by other to the project.>

Note that it is not acceptable to solicit assistance on ‘proof reading’ which is defined as the “the systematic checking and identification of errors in spelling, punctuation, grammar and sentence construction, formatting and layout in the test”; see
`https://www.leeds.ac.uk/secretariat/documents/proof_reading_policy.pdf`

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Algorithms such as Deep Blue [4], AlphaZero [14] and more recently Player of Games[13] have enabled computers to out smart the smartest humans at the game of Chess. Unfortunately all these algorithms are bound to the digital world, rendered useless when competing against humans on a real board. This project aims to explore a major component of this: vision.

Unlike humans, the hard part of chess for computer is not planning which move to take next, but instead recognition, localisation and manipulation of objects in 3D space which currently all present much greater challenges. Perhaps the reason for the vision problem feeling so apparently effortless to humans is that over half of the human cortex is allocated to visual processing [15]. It is also described by Szeliski as an inverse problem and to attribute that as the reason for it's difficulty within computer science [16].

Consider the vision problem for chess to be two-fold: what is the current board state and where are all of the pieces? In particular this project will focus on the former, that is, to produce and present a solution for determining the state of a chess board from a video stream. A solution reliable enough to live up to the likes of AlphaZero in a robotic system, but also a solution that could be immediately useful in other applications such as realtime chess analysis from a real board.

There will be a focus on deep learning techniques, with consideration for best practice and the aim to share the tools to more easily manage and create new datasets in this area. Something called for by [8] as a serious challenge and priority for future research.

1.2 Literature Review

1.2.1 A *Brief* History of Computer Vision

Computer vision is the study of making sense from visual data. Applications include character recognition for digitising documents, segmenting and classifying object instances for cancer screening, object tracking for sports analysis from video and countless more. To reach these goals, we name *features* to be useful pieces of information within an image (or any other visual data) and *feature detection* to be the class of algorithms that can extract features from images.

Some of the earliest work in computer vision started with Larry Robert in his 1965 paper [] describing a simple 2x2 convolutional kernel for edge detection which soon became the predecessor to the Sobel operator in 1968 [] and the still widely used Canny edge detector developed in 1986 []. The ability to find edges provided a helpful backbone for detection of higher order features such as lines and corners.

Another one of these tools that is still commonly used to help more sophisticated feature extractors is *thresholding* which refers to the process of partitioning images into two sub regions based on a pixel color/intensity threshold value. More complex partitioning schemes are often referred to as image segmentation techniques. Otsu's method is a famous example of an automatic global thresholding algorithm that finds a suitable threshold value for the entire image without supervision by minimising the resulting variance both between and within the two separated sub regions [].

For higher order features more sophisticated methods are needed, a bucket term used to describe some of these methods is *template matching*, which can generally be considered as comparing an image to known feature templates to determine if that feature is present[1]. The Hough transform is the one of these higher level feature extractors commonly used to extract lines, circles and even arbitrary shapes, which heavily relies of edge detection [1].

Feature Descriptors (keypoint, intest detectors) ... Laplacian, Harris, SIFT, SURF, ORB, HOG

Neural Networks

Enter... Neural Networks now do all of the above, better. Brief overview of some of the big ones then focus in on a few specific architectures for image classification used throught the project

1.2.2 Computer Vision for Chess

Despite chess being a very narrow application of computer vision, the amount of research effort gone into the problem of determining board state is not insignificant. A variety of approaches have been tried and tested for which the following section will now summarise.

As in [8] the vision problem can be split into two further problems for analysis: board detection and piece recognition.

Board Detection

The problem of board detection is not specific to chess but also receives heavy research from other applications such as camera calibration [7]. The built in camera calibration functions in opencv [3] and matlab [12] are used in many previous works [11, 2] which provide a quick and precise solution for board detection, but becomes unusable when any artifacts like chess pieces are present on the board. This forced those authors to take the approach of an initial setup stage at inference, making the solution unfit to changes in board position during inference.

Due to a chessboard's simple features many early works of line and corner point detection can be applied. For example Hough transforms are used to detect the lines of a chessboard [10, 5]. Corner point detection methods such as the Harris and Stephens's [] were also common among solutions [], with some authors combining approaches with further processing such as canny edge detection [] to yield more reliable results.

ChESS was another corner detection algorithm that out performed the Harris and Stephen's algorithm []. This was, perhaps interestingly, created for real-time measurement of lung

[1] Neural networks perform template matching on an input against their learned internal features

function in humans, further demonstrating the attention chess board detection has received due to its general applicability.

There are many other algorithms that require simplifications such as custom green and red chessboards [], multiple camera angles [], or even the requirement of user input for entering the corners of the chessboard [].

The most impressive work came out of Poznan University of Technology which proposes many interesting ideas that perform more reliably in a wider range of difficult situation such as pieces being present on the board []. They employ an iterative approach with each iteration containing 3 sub-tasks: line segment detection, lattice point search and chessboard position search. In each iteration of line detection a canny lines detector [] is used on many preprocessed variations of the input image to maximize the number of relevant line detections which are then merged using a linking function. The lattice point search starts with the intersection of all merged lines as input, converting these intersection points to a 21x21 pixel binary image of the surrounding area and runs them through a classifier to remove outliers. The addition of a neural network as a classifier greatly improves the generality of the proposed solution as it can be resistant to lattice points that are partially covered by a chess piece. The final sub-task then creates a heatmap over the original image representing the likelihood of a chessboard being present. Under the hood this is done by calculating a polyscore for the set of most likely quadrilaterals formed by the lines of the first stage. The polyscore is a function of the area of the quadrilateral and the lattice points contained within it. It is the quadrilateral that produces the highest polyscore that is used to segment the image for input to the next iteration until the quadrilateral points converge. The main disadvantage of this approach compared to others is that it can take up to 5 seconds to process one frame, for most use cases however this will be sufficient, unless realtime board training is required.

Piece Recognition

Piece recognition has proved more difficult []. Most chess vision systems avoid classifying pieces by type (knight, king, ect.) all together []. These approaches typically get around this by requiring the board to start in a known position. From this known state the normal rules of chess can be used to infer what pieces are where after each move. Simpler methods require human input to prompt when a move has been taken [], more sophisticated attempt to do this move detection automatically.

These automatic move detection methods tend to all follow the same overarching processes of thresholding to detect a move having occurred, whether on color [] or even the edges detected within each square []. Most authors recognise the dependance this approach has on lighting variations, with Otsu thresholding [] sometimes being used to minimise the negative impact when lighting changed. While this improved results for what may be considered normal lighting conditions, they still suffered. They calculate reference colors for all 4 variations (white square, black square, white piece, black piece). All of these then only work in situation where a series of moves are to be recorded according to typical chess rules and not the chessboard state at any given moment.

There were a couple of methods that stood out from the rest each in their own way. One used fourier descriptors to model piece types from a training set and the other modelled the pieces in Blender, a 3D modelling software, utilising template matching to determine piece type. The fourier method was very sensitive to change in camera angles, preferring a side view angle that unfortunately cause too many occlusions to be practical. The template matching approach took over 150 seconds on average to predict board state from one image which does not lend itself to interactive play in a robotic environment.

Go to standford dude and the heatmap guys as the best approach out there. They use SIFT and hard coded color alogrithms. heatmap guys improved on this only by adding more restrictions by assuming the board much be valid and making statistical assumptions on what state is most likely. Oh and a HOG method. The one that said SIFT didn't work well because the lack of texture.

More recently another group of methods have surfaced using neural networks, specifically convolutional neural networks (CNNs) []. One of these used a pretrained Inception-ResNet-v2 model [] and only had 6 classes, resorting to the more tradition approaches for color detection, in particular binary thresholding with added morphological transformations to reduce noise as seen in previous works []. Interestingly the six chosen classes were 'empty', 'pawn', 'knight', 'bishop', 'rook' and 'king_or_queen' as they claim kings and queens can be difficult even for human eyes to distinguish. Because of the choice of classes this method falls back to relying on a chess engine to determine piece type, which while usually correct for normal games of play makes the method unusable for games played with a variation on the normal rules of play. The other two methods used a simpler CNN structure similar to that of VGG [] with 13 classes, one for every colored piece as well as the empty square.

1.2.3 Prior Work From the Author

Mention robotic arm and vision system for two counter board games as well as automatic differentiation library.

Chapter 2

Methods

To summarise, the overall approach for determining board state is to first segment each square of the board from the image, and use those subimages as input to a piece classifier from which the output can be concatenated into a view of the whole board. Splitting the input to a classifier up by square is a unanimous decision taken by previous authors discussed in Literature Review as it reduces the problem to a simple classification problem of only piece type and can take advantage of the thoroughly researched area of chessboard detection.

2.1 Data Collection

At the heart of any machine learning project is the data. It is as important, often more important, than the code and presents many interesting challenges. **Why is this?** Discussed in the following sections are some of the challenges and decisions that were considered.

2.1.1 Sensors

The eminent challenge is acquiring data in the first place and is highly context dependant,

For vision there are a range of sensors we can use to gather data from the real world. Sensor choice is an important choice for any robotics application as there are important tradeoffs, as with any engineering challenge, which must be considered.

Outline some of the tradeoffs between spatial sensors

One important distinction to make is the difference between training and inference. Requirements at the time of training my differ significantly to the requirements at inference. Processing power, energy supply and realtime operation are some of the constraints that will have to be met when considering different sensors.

Talk about single camera.

The sensor used throughout this project is the RealSense SR305 which is a RGB-D camera using structured and coded light to determine depth, it functions best indoors or in a controlled lighting situation. For the reasons outlined above the RGB camera stream is mainly relied upon but there will be some discussion and comparision of piece detection with the depth sensor.

Talk about the generic camera

2.1.2 Auto-Labelling

Talk about using a simulator.

A closely related challenge of acquiring the data is that of labelling it too. During Literature Review multiple past authors have stated the availability of datasets for chess piece recognition is sparse [] with some emphasizing dataset collection took the large majority of their time []. It is also widely known that neural networks scale with the number of examples [], which will be explicitly explored for Chess Vision in Results. This however poses the question: how do we get access to a lot of labelled data for chess?

Unlike techniques in [] **Some examples of other auto labelling techniques** the approach taken here was to maximize speed of collection and flexibility.

Portable Game Notation (PGN) is a common format for recording chess games as a series of moves and are widely available online. All the PGN files used in the project were used from [] which has over one million games. Utilising this data not only has the benefit of an abundance of chess games but also that the recorded games are real and contain positions more likely to appear in game play.

A program is developed for recording these games with the generic camera interface as previously described. The program takes screenshots upon user input (with the [Enter] shortcut) displaying the move number and image as a result for visual feedback before saving to disk. These games can then be automatically labelled using the matching PGN file.

After the development of this pipeline it was possible to collect over 2,500 *unique* labelled images in under two hours.

2.1.3 Dataset Versioning

With all this data the next challenge becomes self evident. It is concerned with the question: How do we manage all of this? As experiments are carried out and iterations on the dataset are performed there will be many changes and variations of the data that are used to train models. This is a challenge with data for many reasons, the first being reproducibility. Say you train a model that performs really well, then you make some changes in say the balancing of the dataset or even the size and you train again. If you go through a series of changes like this and discover that it performed a lot better the way you had it previously then you might want to roll back.

In typical software engineering a version control system like git [] would be used. While this is what was used to version the codebase of this project, there are different challenges with data, namely it's storage. Code usually takes up megabytes of storage, the Linux Kernel source code for example contains 27.8 million lines and is only around 1GB in size []. The data used throughout this project came to >20GB.

There are many proposed solutions for this problem, from managed service like Neptune [] and wandb [], to self hosted opensource options. In fact Git has its own solution called Git Large File Storage (Git LFS) [] which uses the exact same methodology as with code except it will store large files in an external remote and only reference those locations in your git repo using a hash of your data's content. This means if you change any of your data a new copy of that data will be stored and it's reference updated. And so to maintain git's methodology of versioning any change of your data will mean a new copy will be stored, rapidly increasing storage costs with changes. Neptune instead is a more holistic machine learning platform that provides a whole bunch of features other than just dataset versioning. As with other managed services these managed solutions are prone to lock-in and require you modifying your code with a bunch of API calls to get them running. In the end it was found that a custom solution utilising some cloud based object storage proved most useful, with the least amount of effort and cost. Perhaps the wide variety of solutions all with different approaches is evidence of this not yet being a solved problem. To demonstrate the cost difference, Git LFS per GB cost is $0.1 a month which is 4 times more expensive than amazons most expensive rate of $0.023 per GB. With optimisations you can get the amazon S3 bucket price down to $0.0125 making Git LFS 8 times more expensive. These numbers may look small but they add up with time and scale, especially when versioning many changes where copies and diffs need to also be stored.

The solution used in this projects centers around 3 elements: The *Game*, a *Labeller* and the *Storage* facade.

The storage facade's purpose is to abstract file storage so that it could work with any backend and provide a simple to use interface for fetching files using a file system.
`file = open(Storage("img.png"))` for example will give you the file descriptor for img.png cached from the filesystem if it exists, pulling it from an external store if not. This was very useful for training across different VMs in the cloud and could work with any storage implementation. The only backend implemented was Amazon S3 bucket store, and used less than 50 lines of code.

The Game class **??** is how chess data specifically is managed. Each instance of Game represents one 'unit' of data where each unit is a recorded chess game. You can record a game by using the recorder application described in the autolabelling section and is simply a sequence of images with an associating PGN description. *LabelOptions* can be set on a game which will be used by a Labeller to describe exactly how the autolabeller should function. For example, `Game("Kasparov", LabelOptions(margin=50))` will create a game that should be labelled with a margin of 50px surrounding each square. As will be discussed later it is this game object that will version the data used to train a model while dramatically decreasing the storage cost. This is opposed to storing a new entire labelled dataset everytime a slight change is made, i.e chaning the margin of each square.

As different model architectures were explored, different labelled data was need entirely. For example, one model may only be predicting whether a piece is a black or white pieces whereas another may be prediting the type of piece. To cater for this variance while keeping the rest of the training pipeline unchanged the Labeller abstraction is used. Each labeller has a function that can take in a Game and output a series of X, Y input, target pairs. These can then be saved to disk for the *ChessFolder* pytorch dataset to handle. This could even be extended to more complicated labels like bounding boxes or multiple input images.

One important note in implementation is the use of python generators which dramatically reduce memory usage and speed up the auto-labelling pipeline by over 10 fold in places.

2.2 Model Training

Given this data what is are model meant to do.

2.2.1 Experiment Tracking

Over 500 hours was spent on training models with many different architectures each trained hundreds of times on slightly varied hyperparameters. This makes the processes which is commonly refered to as Experiment Tracking a very important one. It would be good to visualise all these experiemnts and be able to roll back to old models. As with dataset versioning there are a plethora of new managed services that promise they can do this for you. The appoach opted for during this project utilised an opensource solution called Guild []. The main reason for this choice was beacause of it's unopinionated nature, requiring zero code changes. GuildAI leverages the filesystem to save *runs* including the environment configuration at the time of training. A run is made from the command line (i.e. `guild run train --remote ec2 EPOCHS=10 LR=0.001`) and represents one iteration of a model and can even be executed remotely via ssh to give easy access to GPU machines. It can also perform hyperparameter optimization using built in techniques like grid search, random search, gradient boosted regression trees or even your own custom optimizer. To enable the zero code change promise, guild fetches hyperparameters from a variety of places including globals set within your training script, configuration files and commandline arguements. This method was greatly prefered to using a paid managed service or some other solution which requires a database to setup. The previously mentioned dataset versioning method also greatly benefits from guilds use of the filesystem as within each run we can also store the Games and Labeller that was used to train the model. TesnorBoard integration also played a huge roll as it provided an easy to use, visual place to interpret and compare runs. See Figure 2.1 for examples.

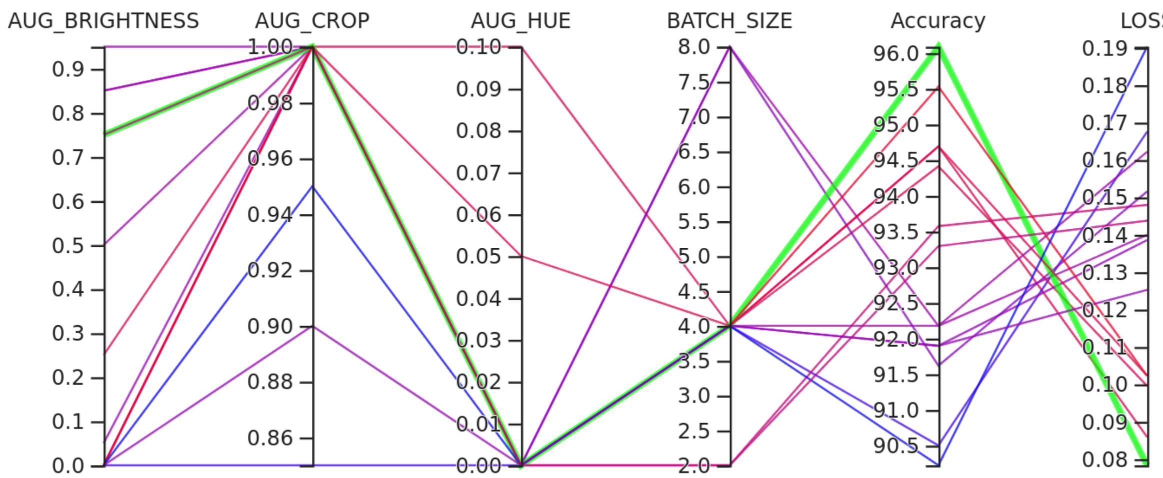
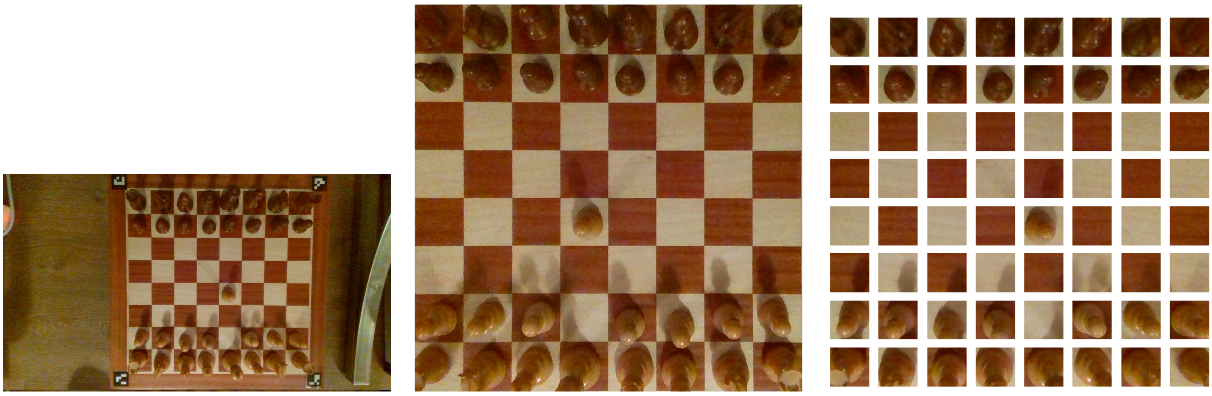


Figure 2.1: A parallel coordinate view of runs within TensorBoard

2.2.2 Board Segmentation

Aruco markers are chosen for board corner point detection as a very simple method that works reliably even with a board full of pieces. With the corner points of the board the perspective transformation is calculated using Gaussian elimination [?], as demonstrated in Figure 2.2. Although Aruco markers require customizing the environment they are very fast at inference and allow the focus to remain on piece recognition, from which Literature Review showed is less studied and reliable. In a more holistic solution the iterative heatmap approach proposed in [6] would be recommended.



(a) Original Image (b) Reorthogonalisation (c) Square Segmentation

Figure 2.2: Square segmentation process. The 64 images in (c), after augmentation, is what ultimateley gets sent to the model

2.2.3 Piece Recognition

Now that the board has been segmented including all of it's squares, it is time for the fun stuff. That is to determine what piece, if any, occupy each square. To start, a good baseline is found. A good baseline is a simple model to understand and easy to get decent results with. For classification, the pathological baseline would be a uniformly random model, which could easily be extended to use a categorical distribution. The probability mass function for a categorical distribution of k categories numerically labelled $0, ..., k$ is

$$f(x \mid \boldsymbol{p}) = \prod_{i=0}^{k} p_i^{[x=i]}$$

where $\boldsymbol{p} = (p_0, ..., p_k)$ and p_i represents the probability of an image of category k being sampled from the training set. Since $\sum_k p_k = 1$ you can generate the probabilities by normalising the count of each category in the training set. Algorithmically sampling from the distribution can be done using inversion sampling which requires calculating the cumulative distribution function.

Of course you could just use pytorch `Categorical(tensor([0.25, 0.5, 0.25])).sample()` []

This is common practice in exploratory machine learning [] so that the transitions made are always from a known and working state. It becomes very easy to see if an experiment is not working by comparing it to your baseline and makes it easy to go back.

Keeping to this strategy, the Multilayer Perception (MLP) or fully connected network [] was the first neural network to be explored. We explore nerual network approaches as they have proved most effective for image classification [], they are also relatively simple to deal with as no hand-crafted feature extraction is needed and end-to-end solutions are much more viable. By starting with the MLP, all complexity from the network is stripped away so the more extraneous elements such as the training loop and evaluation metrics can be built and tested.

Evaluation metrics were very imporatnt in developing as loss is not that human interpretable. Firstly accuracy, which is much more interpreatable, is calculated to be the ratio of correct classifications to total number of sample images. This could then be extended to top-n accuracy which takes accuracy to be:

INSERT MATH.

As the number of evaluation metrics grew it became prudent to encapsulate them into an *Interpreter* class so as to not clog up our training loop which itself is within a *Trainer* class to encapsulate it's implementation away from the training script which will see frequent changes during experimentation. Some of the first additional methods added to the Interpreter is `plot_top_losses` and `plot_confusion_matrix` to make it significantly easier to find which classes the model was confusing and which specific images it found difficult to classify giving hugely helpful insight into the dataset itself and even finding bugs in the autolabelling pipeline.

Before training, a fixed random seed was important so that it was possible to tell whether or not any particular change was positive since all models started from the same initalisation. The initalisation of the final layer was hard coded to be an equal distribution, sensible initalisations of final layer can have a noticable effect on training convergence [].

SHOW PLOTS

Once the full training and evaluation structure is functional, new features can be incrementally added and architectures explored. - human labelled dataset?

Another strategy was to purposely overfit models during training. It was easy to tell when a model was overfitting by splitting the data into training and validation sets [] and viewing the relative losses in TensorBoard as can be seen in **??**. The reason for doing this was that if the model was able to overfit (i.e. achieve $\sim 100\%$ accuracy on the training set) then it was able and big enough to learn distinguishing features or as is sometimes put: the model has sufficient entropic capacity. It is then a lot easier to make it generalise well, for example by reducing the number of parameters, than to go from an underfitting model to a generalised one as you don't necessary know where the problem lies for not fitting to the data.

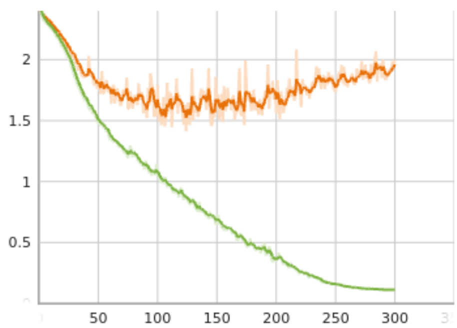


Figure 2.3: Demonstration of overfitting. Loss against epoch. The validation loss in orange is seen diverging above and the training loss in green is seen tending to zero below2. A model should be saved at roughly epoch 125.

There are many ways to fight overfitting as typically referered to as regularization techniques, perhaps the easiest of which is early stopping []. The way this was implemented in our Trainer was to save the model after the first epoch and consquently after every other epoch for which the validation loss was less than the previously saved model. It is possible to then stop training entirely if after a set number of iterations no improvement is seen in the validation loss.

As mentioned in Literature Review convolution operations, and in particular differentiable convolutional operations have had monumental impact on the field of computer vision and so this was the next experiment. Quite quickly, especially with a limited dataset, overfitting became a major problem to overcome and so many experiments were positioned to solve this problem. Below are an overview of some of those experiments including some final optimisations to squeeze as much performance out of our model as possible.

Architecture

ConvNext https://arxiv.org/pdf/2201.03545.pdf

Due to the recent surge of deep learning research for computer vision there are a huge range of openly available models. During this project ResNet, ConvNext, and ViTs were experimented with. How much detail should I go into here?

ResNet

Loss Function

https://towardsdatascience.com/choosing-and-customizing-loss-functions-for-image-processing-a0e4bf665b0a https://arxiv.org/pdf/2009.13935.pdf proposes SML may be better than cross-entropy

Optimizer

[put into results] AdamW was found produce more attractive results from comparing SGD with/without momentum and Adam and AdamW [] Learning Rate, Weight Regularization.

Learning rate scheduler, leave this till the end.

Batch Normalization

https://arxiv.org/abs/1502.03167 and renorm for small batch sizes https://arxiv.org/abs/1702.03275

https://arxiv.org/abs/1207.0580 shouldn't be used after convolution layers [] there has been some work https://arxiv.org/abs/1904.03392

Pooling and DropPaths

Transfer Learning

There appears to be a trend occuring in the deep learning space. Some organisation spends millions training an impossibly large nerual network and others more and more are using these models, often fine-tuning for their own use cases. [] uses these large models as fixed feature extractors.

Labeller	Count	Classes
all	13	Empty, White Pawn, White Rook, ..., Black Queen, Black King
piece	12	White Pawn, White Rook, ..., Black Queen, Black King
occupied	2	Empty, Occupied
color	2	White, Black
type	6	Pawn, Rook, Knight, Bishop, Queen, King
type+	7	Empty, Pawn, Rook, Knight, Bishop, Queen, King

Figure 2.4: Class Sets

This appraoch makes sense as it is inpractical to retrain huge nerual networks that take weeks, millions of dollars and wasteful amounts of energy to train [].

In the case of CNNs we can see the features that kernals in the early layers learn [] are often very simple shapes and will be common for all computer vision tasks. This will explored this further in the results section as we visualise kernals from both random initalised models and pretrained models.

Ensemble and Multitask Learning

Another regularization technique is known as ensemble learning and is common practice in deep learning to reduce variance inherent to stochastic training with random initalisation [9]. The technique involves training several models (with different random seeds or configurations) and combining their outputs. Using an ensemble almost always beats training a single model as two models are unlikely to make exactly the same errors on an unseen dataset. This projects extends on this idea to also combine models with different class predictions.

In the two previous deep learning solutions for chess piece recognition, neither researched into the effect of using differnt class groups, despite chosing differently. In this work, 5 sets of classes were considered as shown in Figure 2.4. Models trained on each set of classes were compared with accross each metric and some were selected to be used together. For example, combining the predicitions of 3 model separately trained with the 'piece', 'color' and 'occupied' labellers would yeild the same class of predicitions as in 'all'.

Every model you add to an ensemble has a large impact on the computational cost at inference (as well as training) which must be considered. A separate approach called mutlitask learning which separates heads for different classes in a network might be worth considering instead. Part of the benefit of separating models for different classes (which can be refered to as tasks in this context) is that they each get their own parameters they can optimise. By using 'all', each task has to fight for optimisation. Some tasks may train well together and benefit from sharing parameters, some may not. Multitask learning is the study of finding this balance of which parameters should be shared amoung each task.

It is also worth mentioning loss functions here as they will need some attention if multitask learning is to be used. Typically a separate loss function will be used for each task and then these are simply summed together before performing backpropogation. One useful technique is to be able to weight these losses towards the more difficult tasks. This weight vector then becomes another hyperparameter to optimise. It's good to know that this technique can used with many individual loss functions across specific classes too.

Augmentation

Data augmentation is strategy every machine learning practioner has in their aresonal. It addresses the data problem, allows us to truly leverage the data we have and generalise our models further. The MINST [] dataset itself was created using data augmentation. In the case of chess piece classification, the correct augmentations or transformations should be chosen. Go onto list the transformations used and why.

2.2.4 The Final Model

Augmentation as well? Talk about grayscale (a lot of previous works used gray scale do performant CNNs aid from gray or hinder?)

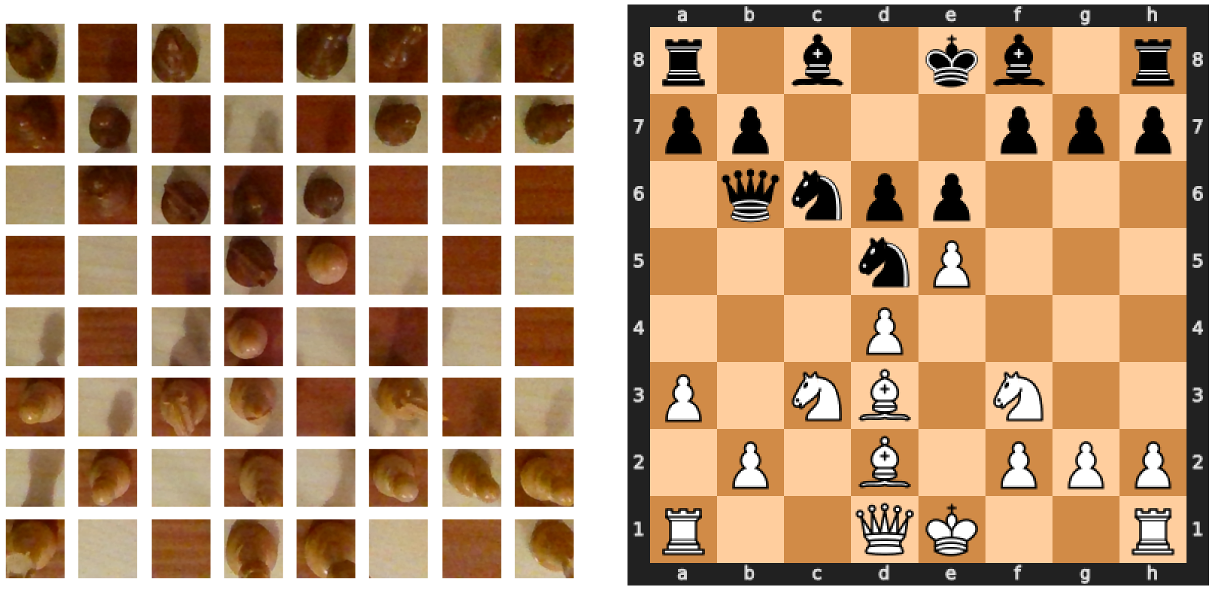
2.3 Recording a Chess Game to PGN

The goal of this section is to discuss methods for using the proposed piece classifier to record an entire game of chess, played on a real board, to a PGN formatted file. This application is later referered to as the *Inference Application*.

The general approach is to generate a board state at each fetched frame. That is to segment the board and each of it's squares, send each square through a forward pass of the piece classifier described above and finally collate the predictions together into a board state. A board state is a generic term that in actuality could be many things, but in this case it is sufficient enough to think of it as Forsyth-Edwards Notation (FEN). This board state can then be compared with the previous board state and if any difference is detected then that move is added as a child node. This tree structure gives flexibility for many variations of the same game to be recorded to later be parsed and encoded to PGN.

2.3.1 Leveraging a Chess Engine

As leveraged by others before [] a chess engine and other statistical methods [] can be used to increase the reliability of board state prediction when the normal rules of chess can be assumed to be abided by. The proposed piece classifier in 2.2.4 makes no such assumptions, which lends it self to a wider array of circumstances even when the normal rules of chess are not being used. When a game is to be recorded to a PGN file however there are some rules you must follow so that a game can be encoded. Typically chess engines can handle a few variants and so it is assumed to be safe to incorporate one here without loss of too much generality. And so a chess engine is used, not to help the classifier's predicitions, but to automatically determine when a move has been made. Specifically leveraging a chess engine means that we assume (or set) the starting state of the board and only allow legal moves within a chosen chess variant to be accepted, much like a human would when playing against a competitor.



(a) Frame after Segmentation (b) Prediction Vision State

Figure 2.5: Board Squares Segmentation Process

To do this, two concepts are introduced: *VisionState* and *BoardState*. With some simplification, the VisionState is a lightweight representation of board as the model sees it in the last fetched frame. The BoardState starts at the known starting state and is only updated if the VisionState at any time step represents a legal move from the current BoardState. When the BoardState reaches a terminal state, or otherwise receives a cancel signal in cases of a draw or resignation, the game tree is parsed and the PGN saved to disk.

Since our classifier is almost guarenteed to never achieve 100% accuracy on new unseen data, especially as more chess sets of different shapes and sizes are used, it is likely to have varying uncertaintly. Often, the more uncertain predictions of a particular square may result in flickering, that is for example it deduces the piece to be white in one frame yet black in the next. This occurs when the resulting probability distribution for a given input looks something like $p = [0.38, 0.39, 0.09, 0.04, 0.1]$ as it is not unlikely with the next frame we see it change to $p = [0.39, 0.38, 0.09, 0.04, 0.1]$ resulting in a different classification. To cater for this uncertainty like this, the VisionState also has a concept of memory with length N, where memory is in an average of states over the last N frames.

2.3.2 Motion Detection

One factor that was found to still sometimes break this system was motion. Moving pieces across squares and hands flailing over the board confused the model. Even with the separation of the BoardState and memory to remove anomalous predictions - especially when motion persisted for longer periods.

In a lot of these situations the actual board state in undefined. That is because a piece has been lifted and so most now be moved but not yet let go and so it's definition may be undetermined. Because of this is it not unreasonable to halt inference all together. User input to indicate when a move has been completed is a common strategy [], but goes against the purpose of building such a system all together - autonomy. Instead a motion detector is employed. Even the naive motion detector of using a threshold over the absolute difference between each consecutive frame was found to be sufficient for removing these disturbances. Some other methods such as SIFT and SURF were also explored but found to be more computationally expensive that necessary.

2.3.3 Towards Continual Learning

Another feature added to the Inference Application was the ability to save snapshots. During inference it was not uncommon to see the vision state make mistakes, and while the above features usually enabled us to generate an accurate PGN file, this is still useful data. By giving the user the ability to save snapshots when they see an error it is now possible to feed this data back into fine-tuning the model. This is the process of continual learning and is an active area of research [].

Chapter 3

Results

This chapter will cover the results of the best and final model that was trained. The first section, Piece Recognition, measures the performance of piece recognition against existing solutions. The second section, Recording PGN, tests the model within the inference application to give a better view of its utility.

3.1 Piece Recognition

The best openly available solutions for chess piece recognition were found and trained on the chessboard used throughout this project. As can be seen, the presented solution outperformed the others by a huge margin, despite using data from the evaluation set that had never been used to train the presented model. In order to make a fairer comparison another board entirely was also chosen for these final tests. The reason being that it is more than possible through each iteration and optimisation of hyperparameters the presented solution was being overfit to the images in the evaluation set despite never being directly trained on it. By including a never seen before dataset more meaningful conclusions regarding the generality of the model can be drawn.

[] Uses a pretrained VGG16, freezing all layers of the convolutional feature extractor and adds a head of 3 fully connected layers with a total of 2.5M parameters. The final layer outputs a softmax distribution over all 13 class ('all' from Figure 2.4). After getting familiar with the code base and running a few experiments with different datasets, a few minor improvements were spotted that could be made without changing the architecture, such as implementing early stopping. In order to ensure a fair representation of the authors work those changes were implemented and the best results were used in Figure 3.1.

[] Trains a Support Vector Machine classifier for each piece type including empty ('type+' from Figure 2.4) using the HOG features extracted from each training image. To determine piece color they use reference colors which were calculated separately for each board, which unfortunately made this method not easily generalizable when using multiple boards and so the decision was made to exclude it from the final test.

	Chessboard 1		Chessboard 2		Both 1 & 2	
Method	Accuracy	Balanced	Accuracy	Balanced	Weighted	Balanced
proposed	0.97	0.95	**94%**	91%	**94%**	90%
[]	0.87	0.71	0.86	0.75	74%	70%
[]	0.77	0.57	0.74	0.48	-	-

Figure 3.1: **Evaluation Accuracies** for models trained on both single chessboards and then a dataset containing images from both boards

Weighted accuracy is the typicall accuracy calculation:

Notice other metrics such as recall and precision are not included here. This is because there is not

3.1.1 Trials

For a clearer comparison in the task of full chessboard state recognition a methodology is used similar to that of Ding, Czyzeqskil et al. [8, 6]. It involves collecting a small yet diverse benchmark of chessboard images and evaluating the full chessboard prediction of each model. This is in contrast to using the dataset used above which consists only of individual pieces and their corresponding labels.

Screen shots with the results.

table of end-to-end experiments

Talk about speed of inference and any other limitations of both presented and other work.

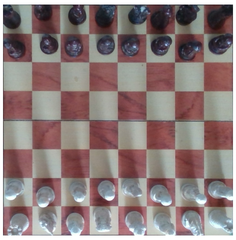
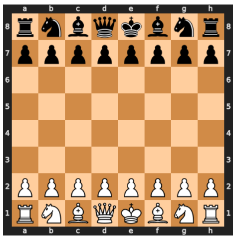
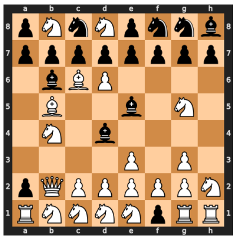
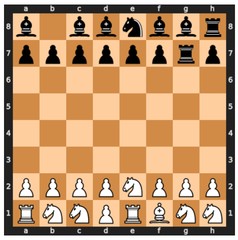
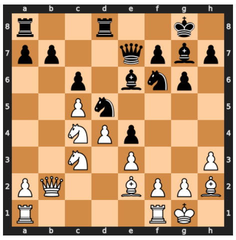
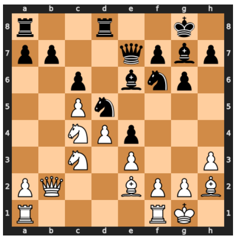
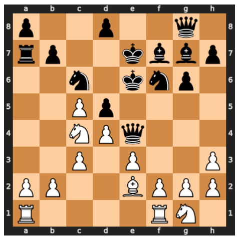
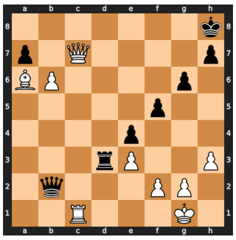
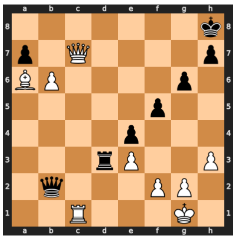
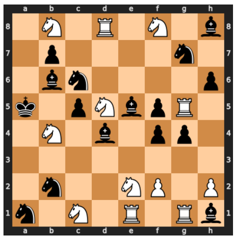
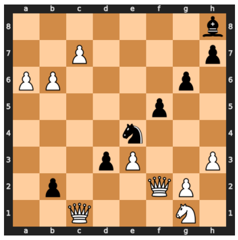
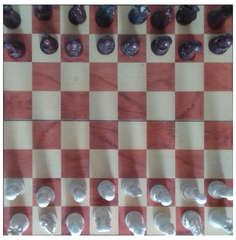
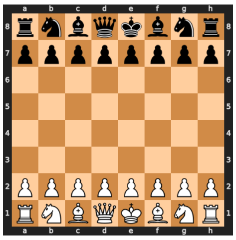
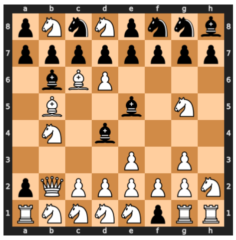
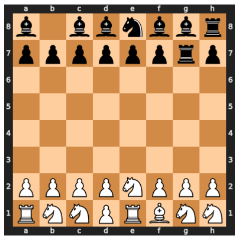
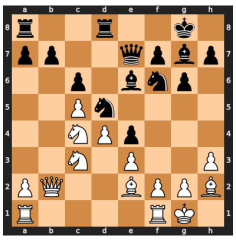
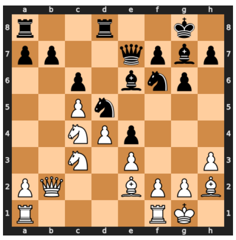
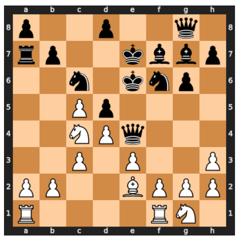
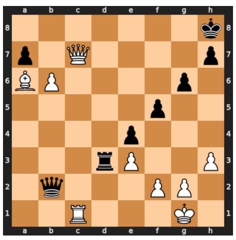
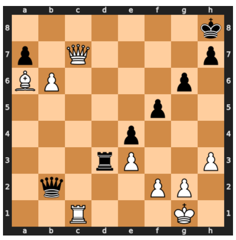
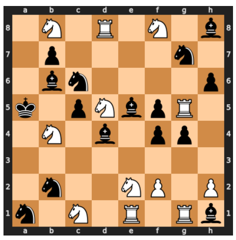
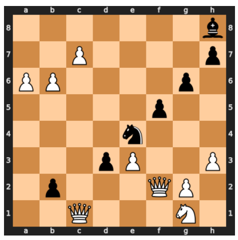
3.2 Recording PGN

In addition to evalutaing the raw model's performance, six games of varying length are played from beginning to end with the inference application. How well does it perform. What did it take to get there?

The one failed game was due to the inference application registering a rook movement before the player lifted their hand from the rook and then decided to move it to a different square.

The fastest move to register took 0.27s and the slowest took 76s which appeared to be when there was lots of specular light reflecting from the pieces causing the model to confuse colors.

How about performance? Frames per second. Analyse profiler.

Input	Ground Truth	Proposed	[]	[]
				
	Mistakes:	0	12	3
				
	Mistakes:	0	12	3
				
	Mistakes:	0	12	3
				
	Mistakes:	0	12	3
				
	Mistakes:	0	12	3
				
	Mistakes:	0	12	3

			Delay for Move to Register (s)		
Game	Total Moves	PGN 100% Correct	Median	Mean	Std Dev
Carlsen/Wesley	57	✓	1.25	3.01	5.26
Carlsen/Rapport	36	✓	1.13	3.21	4.97
Carlsen/Nakamura	71	-	0.95	1.82	2.09
Carlsen/Aranian	55	✓	1.24	4.20	11.31
Fool's Mate	4	✓	0.95	1.19	0.76

Figure 3.2: Games played with the inference application

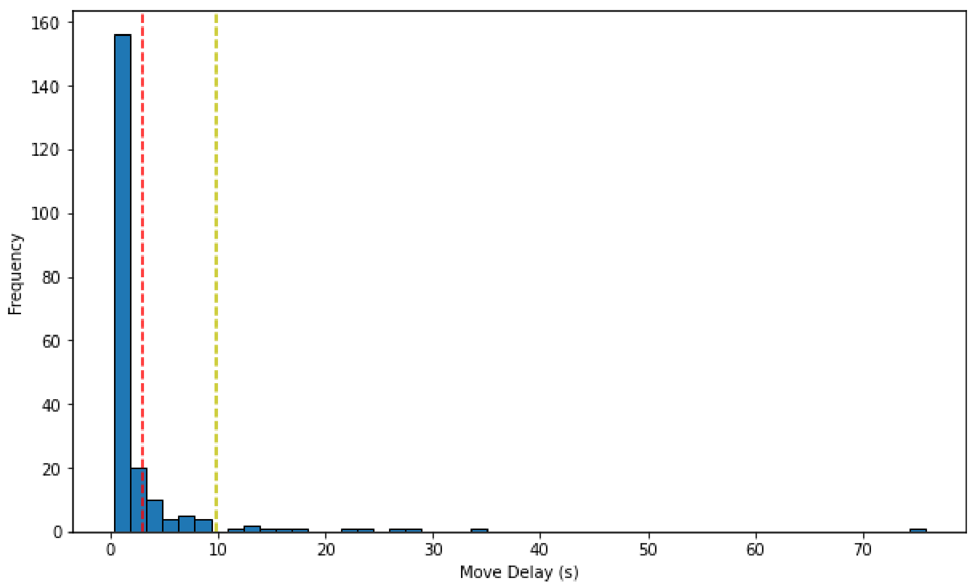


Figure 3.3: Histogram of move delay across all games in Figure 3.2 with arithmetic mean in red and sample standard deviation in yellow

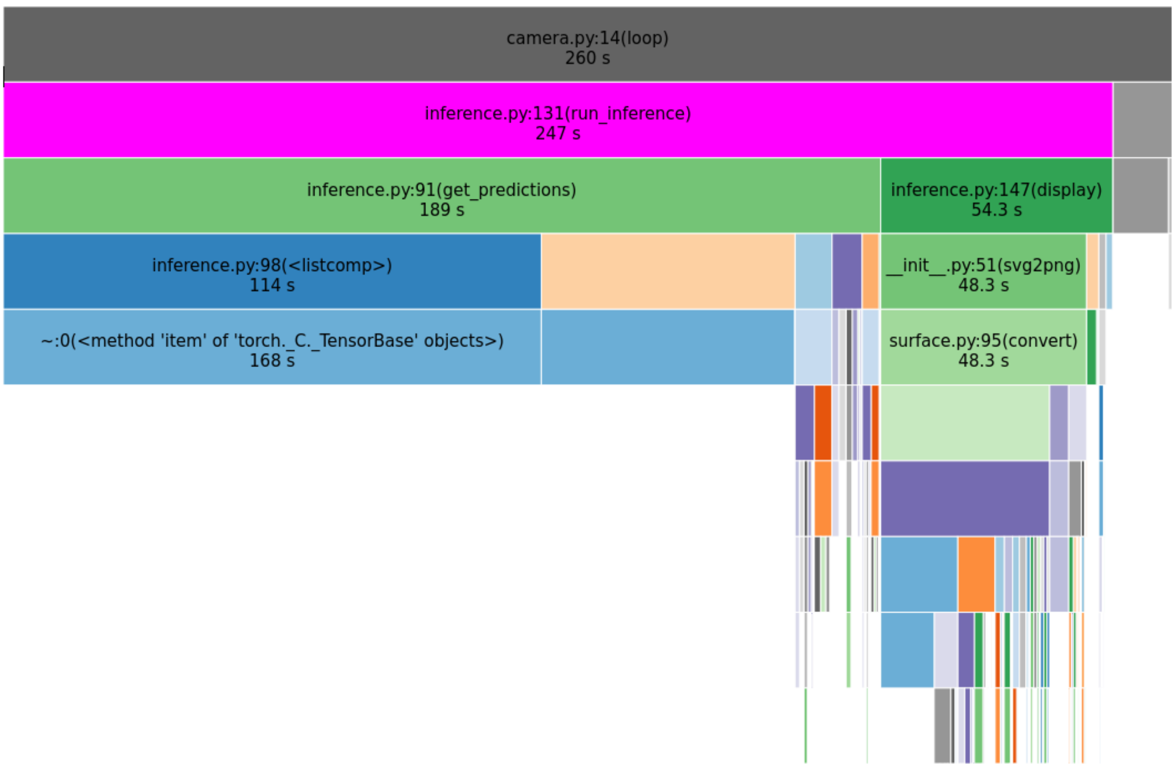


Figure 3.4: cProfile Visualisation of recording a 4 minute game at inference with moves made in quick succession

Chapter 4

Discussion

4.1 Workflow

Throughout development, every change was monitored and evaluated. As expressed in Methods, the use of the Game and Labeller abstractions along with guild for experiment tracking greatly increased speed and ability of evaluate changes.

What was the result of the data labelling pipeline How well did it work What could be improved

4.1.1 Dataset Management

4.2 Board Segmentation

The decision to use aruco markers, while impractical for release, aided development in a couple of ways. Firstly, they're reliable []. To correctly classify pieces, the segmentation of board squares was critical as a small error of even 10mm could completely throw the model off as due to added perspective shift a square could now contain two pieces. Compared to the iterative heatmap method [] aruco marker are not the best, but in a controlled environment this did not become a problem and more importantly they worked even with pieces on the board which enabled recalibration during inference and while collecting data. This is unlike a lot of other proposed solutions []. Secondly they're fast []. The iterative heatmap method can take around 5 seconds to segment the board which while not terrible can add up when relabelling lots of data for many model experiments.

As this project focused on piece recognition and the development of an inference system, aruco markers proved to be a sensible choice. However, they come with deal breaking consequences if this method is to be used in production with many boards. It's just too impractical to print and fix markers every time a new board is to be used with the system and again goes directly against the main purpose of this project: autonomy.

4.3 Inference

Added the ability to take snapshots of misclassification during inference which could be used to fine-tune the model and leads us down the path of continual learning. Outside the scope of this project.

Surprised at how well it worked.

4.3.1 Multitask

Initially a benefit of having the depth sensor is an easier way to detech piece presence. Fixed threshold vs clustering. Adding a margin. How we actually did it. Using paried T-test to evaulate.

4.4 Dataset

I found not only managing datasets but choosing and building them to be a big yet fun task. No doubt by the legends who've built tools for me to use, my experience has been that most of the work is in the data. Creative ways to collect it, label it and fit it to pre-existing network architectures.

It was not until I recorded and labeled lots of data (1000s images) was it until I suddenly started to see new patterns I had not seen before. Small things like dark shadows, motion blur, pieces half across squares, poor resolution or exposured frames.

More control over the sensor hardware sensors may have been useful. Changes in the way we generate image (as in the autolabeller) or adding augmentation such as cropping and blurring helped which these changes that I would not have seen in my metrics before hand.

4.5 Conclusion

4.6 Ideas for future work

Firstly it would be nice to explore these methods with more extreme camera angles. This would probably include extending the labeller too add more margin in one direction to account for the perspective. As my model performs better than [] who found decent performance with other angles, mine should do pretty well, if not better.

If assumptions about the environment are to be kept minimal then localising pieces in 3 dimensional space should be a requirement for robotic manipulation to be possible.

Perhaps worth taking a step back and reconsidering the unanimously made decision by which chessboard state recognition is done. Instead of splitting the board up into squares and using simple image classification it would be more useful for robotic systems to have a 3D representation of the space. It is the author's belief that "3d reconstruction" would kill two birds with one stone and the direction that future research should focus.

Explain. Similar work in other areas.

References

[1] D. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981.

[2] d. bowers. Robot arm, chess computer vision, Jul 2014.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] M. Campbell, A. Hoane, and F. hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[5] W. L. K. Chua Huiyan, Le Vinh. Real-time tracking of board configuration and chess moves. National University of Singapore CS4243, 2007.

[6] M. A. Czyzewski. An extremely efficient chess-board detection for non-trivial photos. *CoRR*, abs/1708.03898, 2017.

[7] A. de la Escalera and J. Armingol. Automatic chessboard detection for intrinsic and extrinsic camera parameter calibration. *Sensors (Basel, Switzerland)*, 10:2027–44, 03 2010.

[8] J. Ding. Chessvision : Chess board and piece recognition. 2016.

[9] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[10] J. Hack and P. Ramakrishnan. Cvchess: Computer vision chess analytics. Stanford CS231A, 2014.

[11] C. Koray and E. Sümer. A computer vision system for chess game tracking. 2016.

[12] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[13] M. Schmid, M. Moravcik, N. Burch, R. Kadlec, J. Davidson, K. Waugh, N. Bard, F. Timbers, M. Lanctot, Z. Holland, et al. Player of games. *arXiv preprint arXiv:2112.03178*, 2021.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[15] R. Snowden, R. Snowden, P. Thompson, and T. Troscianko. *Basic Vision: An Introduction to Visual Perception*. OUP Oxford, 2012.

[16] R. Szeliski. Computer vision algorithms and applications, 2011.

Appendix A

Self-appraisal

<This appendix should contain everything covered by the 'self-appraisal' criterion in the mark scheme. Although there is no length limit for this section, 2—4 pages will normally be sufficient. The format of this section is not prescribed, but you may like to organise your discussion into the following sections and subsections.>

A.1 Critical self-evaluation

A.2 Personal reflection and lessons learned

Surpised at how effective transfer learning is and the significance of it's place in the future. The importance of experiment tracking for research.

A.3 Legal, social, ethical and professional issues

<Refer to each of these issues in turn. If one or more is not relevant to your project, you should still explain *why* you think it was not relevant.>

A.3.1 Legal issues

A.3.2 Social issues

A.3.3 Ethical issues

A.3.4 Professional issues

Appendix B

External Material

<This appendix should provide a brief record of materials used in the solution that are not the student's own work. Such materials might be pieces of codes made available from a research group/company or from the internet, datasets prepared by external users or any preliminary materials/drafts/notes provided by a supervisor. It should be clear what was used as ready-made components and what was developed as part of the project. This appendix should be included even if no external materials were used, in which case a statement to that effect is all that is required.>