School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Chessboard State Prediction with Multitask Learning

Calum Wallbridge

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

2021/22

COMP3932 Synoptic Project

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

Chessboard state prediction is the task of generating a digital representation of a real chessboard from an image. This report details the creation of such a system that outperforms the best openly available solutions. A novel auto labelling pipeline is used to collect a large dataset with minimal effort. Further to this an application is built that is able to successfully record chess games to PGN from a video stream. The proposed solution, auto labelling pipeline and PGN recording application have been made open source and is available for use at https://github.com/Mulac/chess-vision.

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 [5], AlphaZero [38] and more recently Player of Games[37] have enabled computers to outsmart 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 a computer is not planning which move to take next. It is 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 [40]. It is also described by Szeliski as an inverse problem, and attributes that as the reason for it's difficulty within computer science [41].

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; 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 [14] 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 visual data and *feature detection* to be the class of algorithms that can extract features from visual data.

Some of the earliest work in computer vision started with Roberts from his 1965 paper [36] describing a simple 2x2 convolutional kernel for edge detection which soon became the predecessor to the Sobel operator in 1968 [23], and subsequently the still widely used Canny edge detector developed in 1986 [6]. The ability to find edges provided a helpful backbone for detection of higher order features such as lines and corners.

Removing noise from visual data was an important preprocessing step for many algorithms. Gaussian smoothing is the process of blurring an image with the Guassian function and is often

routinely used as a preprocessing step [44]. To note, the difference of Gaussions is also powerful technique for edge detection that is more tolerant to noise than a lot of the other edge detectors [10].

Another tool 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 entire images without supervision, by minimising the resulting variance, both between and within the two separated sub regions [22].

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 a common example of a higher level feature extractor, relying heavily on edge detection, to detect lines, circles and arbitrary shapes [1].

In 1999, David Lowe invented a method that built upon the generalised hough transform and DoG's to detect more complicated features at varying scales [31]. This method is widely known as SIFT, or scale-invariant feature transform, and has become a catalyst for many more approaches that follow a similar approach: RIFT (rotation-invariant feature transform) [27], SURF (speeded up robust features) [2], Gauss-SIFT [29] and many more.

Face detection algorithm. Viola/Jones

Neural Networks

Fast forwarding to 2022 almost all state-of-the-art computer vision solutions are built with deep learning. A field that aims not only to solve vision but intelligence more generally. Deep learning is centered around artificial neural networks which roughly mimic the behavior of neurons within biological brains [16]. Arguably the first neural network applied to computer vision is the neocognitron by Kunihiko Fukushima in 1979 and was used for handwritten character recognition [17]. At Bell Labs Yann LeCun pushed the use of neural networks for real world vision problems forward, applying the backpropogation algorithm and differentiable convolution operators to a neural network like Fukushima's allowing raw images to be used as input to detect hand written digits [28]. This then, in 1989, is largely considered the birth of Convolutional Neural Networks (CNNs) and the beginning of the infamous MNIST dataset. There was another dataset however; ImageNet. ImageNet was the first dataset in computer vision of its size with over 11 million labelled images and 1000+ classes. A competition was held every year starting in 2010 [12], but the best performing methods remained as classical techniques such as SIFT. Two years later, a submission for this competition, AlexNet [25], marked a breakthrough moment for CNNs beating the previous state-of-the-art accuracy by 13%.

CNNs have since progressively improved remaining in the number one position. Most of these methods aimed at trying to solve the vanishing gradients problem [21] and overfitting with

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

techniques like skip connections [20]. This was until the rise of transformers which signified another step change, not only for the field of computer vision but almost any field were neural networks are useful. Transformers doubled down on global attention based mechanisms, dispensing convolutions entirely [43]. Originally made for the problem of machine translation in natural language processing, transformers changed the game, enabling language models such as GPT-3 [35] and BERT [13] that have seen mainstream media attention.

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 [14] 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 [11]. The built in camera calibration functions in opencv [4] and matlab [32] are used in many previous works [24, 3] which provide a quick and precise solution for board detection. However, this becomes unusable when any artifacts like chess pieces are present on the board. This forced these authors to take the approach of an initial setup stage at inference, making the solution unfit to changes in board position during play.

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 [19, 7]. 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 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 situations 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, where the polyscore is a function of the area of said quadrilateral and the number of 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 had less active research than board segmentation. Most chess vision systems completely 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. This of course requires detecting the movement of pieces. 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. They all use thresholding, whether on color [] or even the edges within each square []. Most authors recognise the dependence 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 situations where a series of moves are to be recorded according to typical chess rules, and not the chessboard state at any given moment.

A couple of methods 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 is very sensitive to change in camera angles, preferring a side view angle that unfortunately causes 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.

" Histogram of oriented gradients (HOG) is an example of a higher level feature extractor that

does not perform template matching. Instead it directly describes higher order features as a distribution of intensity gradients. And thus, unlike many other methods, does not benefit from preprocessing in the form of thresholding, blurring or edge detection. "

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 traditional 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. While usually correct for normal games of play, this 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

Previously has built a robotic arm and vision system that can successfully play two counter board games Mention robotic arm and vision system for two counter board games as well as automatic differentiation library.

Chapter 2

Methods

The overall approach for determining board state is to first segment each square of the board from the image, and use those sub-images as input to a piece classifier, from which the output can be concatenated into a view of the whole board. Classifying individual square images 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 because 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.

In order to interact with the camera an abstract `BaseCamera` class was defined that exposes the interface through which any camera can be controlled []. It includes three actual implementations `RealsenseCamera`, `Camera`. The Realsense camera allows depth cameras from Intel to be used and the Camera class can work with almost any RGB camera. The Camera class also let's you use a video file as the source. This is especially useful for debugging and testing as a real camera and chessboard setup is not need.

2.1.2 Auto-Labelling

Talk about using a simulator.

A closely related challenge of acquiring the data is that of labelling it too. Following from the Literature Review, multiple past authors have stated the availability of datasets for chess piece recognition is sparse [14, 9] with some emphasizing dataset collection took the large majority of time [42]. It is also widely known that neural networks scale with the number of examples. However this 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 in this report is to maximize speed of collection and flexibility.

Portable Game Notation (PGN) is a common format for recording chess games as a series of moves. All the PGN files used in this project were obtained from PGN Mentor, 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] key) 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. Consider training a model that performs really well, then you make some changes in the balancing of the dataset and you train again. If a series of changes like this are made and then it is discovered that the model performed a lot better the way it was previously, then the option to roll back would be very useful.

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 visual data, namely it's size. 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 services like Neptune and wandb, to self hosted opensource options. Git has its own solution called Git Large File Storage (Git LFS), which uses the exact same methodology as it does with code, except it will store large files in an external remote. In your git repo, only then are references (made from a hash of the data) stored to the location of those files, instead of the files themselves. This

means if you change any of your data, a new copy of that data will be stored and it's reference updated. 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

2.2.1 Experiment Tracking

Throughout this project, over 2000 models were trained, each slightly different, whether in architecture, hyperparameters or the in the data used. 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.

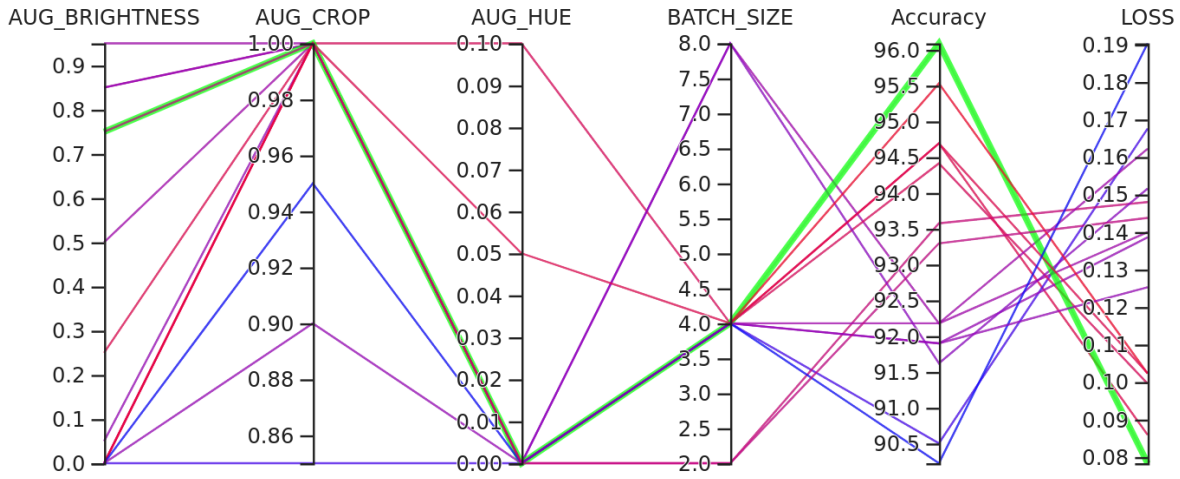
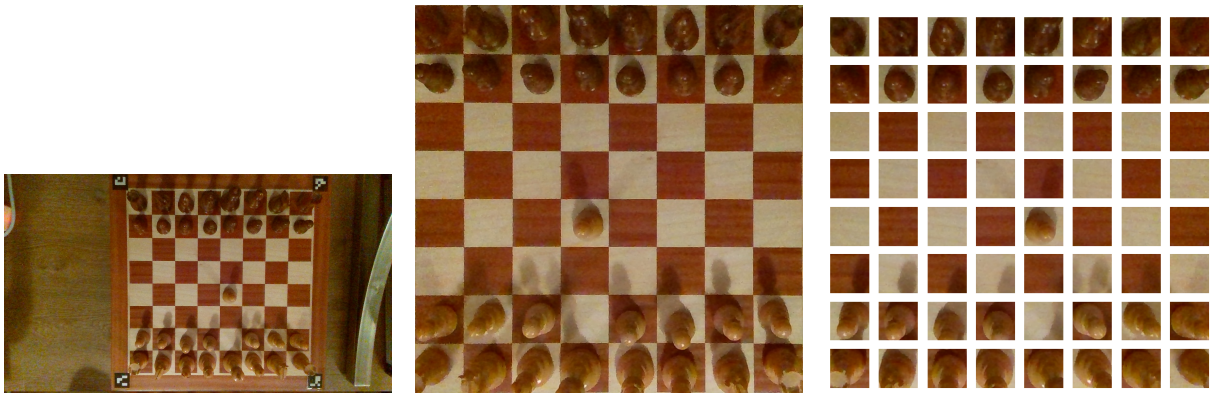


Figure 2.1: A parallel coordinate view of runs within TensorBoard

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 [9] 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 for classification

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.

Starting with a known baseline is common practice in exploratory machine learning [33] so that transitions are always made from a known and working state. It becomes very easy to see if an experiment is not working by comparing it to your baseline.

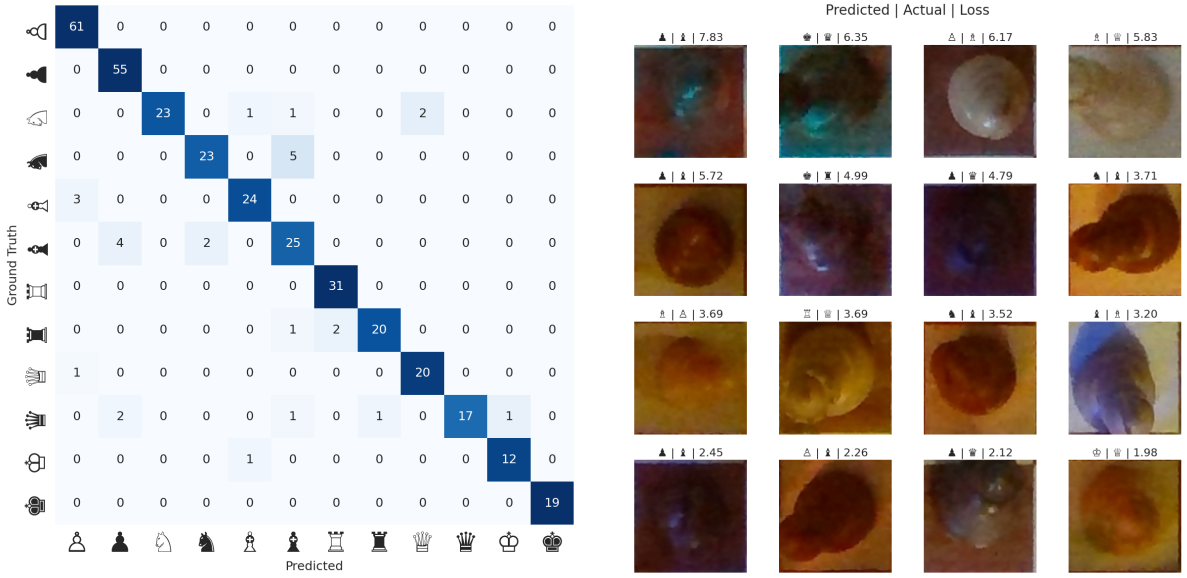
Keeping to this strategy, the Multilayer Perception (MLP) or fully connected network [34] was the first neural network to be explored. We explore nerual network approaches as they have proved most effective for image classification, they are also simple to deal with in the respect that no hand-crafted 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 correct classifications (true positives) to be the count of classifications where the actual class was within the top n confidence scores of the models output. This is useful for problems where there are a large number of classes or when at inference there is the intent to do a search through the output probability distribution which is more relavent to this project as there is at most only 13 classes and a search through the probability space at the inference stage is entirely possible.

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 [26].



(a) Confusion Matrix Example

(b) Top Losses Example

Figure 2.3: Plots that are saved for each run and can be viewed in TensorBoard

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.

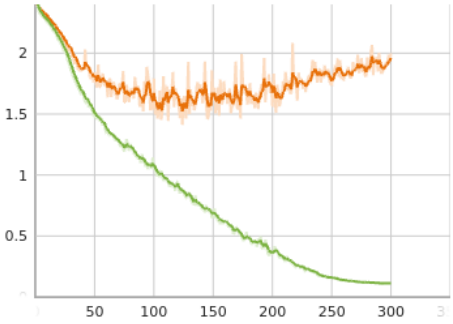


Figure 2.4: Demonstration of overfitting. Loss against epoch. The validation loss in orange is seen diverging above and the training loss in green is seen tending to zero below2. A model should be saved at roughly epoch 125.

There are many ways to fight overfitting as typically referered to as regularization techniques, perhaps the easiest of which is early stopping [45]. 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

After implementing the first fully connected neural network a sequence of gradually more recent architectures were experiemnted with. The first experiment was to reimplemnt Yann LeCunn's LeNet from 3, after this however, it made much more sense to use the already implemented and tested versions in pytorch's vision module. The following additional networks were tested: ResNet [20], ViT [15] and ConvNeXT [30]. In the end it was required to properly understand these implementations so that they could be altered for multitasking which will be discussed shortly.

In the end ConvNeXT was chosen for this project for it's simplicity over ViTs and notable performance increase over ResNets. The ConvNeXT architecture is a pure convolutional neural network and is built up directly from a standard ResNet. The authors of ConvNeXT took an array of design choices from ViTs and applied them to ResNets. Some notable changes that

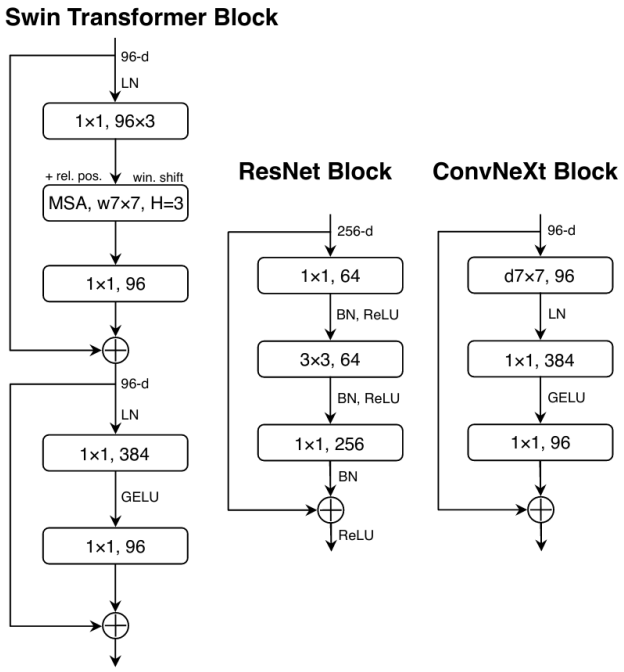


Figure 2.5: A comparision of block designs for Swin Transformer, ResNet and ConvNeXt

improved the performance, can be seen in 2.5 and include larger kernel sizes, few activations (and the use of GELU over ReLU), inverted bottleneck (the fully connected layer is 4 times wider than the input dimension).

Optimizer

Initally stochastic gradient decent with momentum of 0.9 and a learning rate of 2e-4 was used for all experiments. Later when the architecture was found, and larger datasets began to be used, both the Adam and AdamW optimizers were tested. AdamW was found to dramatically speed up learning and even increase accuracy by 0.4%. The final change that was made in this area was the use of a learning rate scheduler, in particular the 1cycle learning rate policy that increases the learning rate from the inital rate before dropping it right down to some minimm towards the final epoch [39].

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.

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.

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 2.6: Class Sets

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 [18]. 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 2.6. 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 often treated as a naive method to squeeze the most possible out of a given dataset that is possible, in fact the MINST dataset itself was created using data

augmentation [8]. While this is true, I have found it move more useful to look at data augmentation as just another regularization strategy to avoid overfitting to your training set. For this reason augmentations were not added until other more architectural decisions were made.

It is important that the correct augmentations are chosen as just throwing any augmentation function at a dataset isn't always going to improve results. For example, in the case here of chess piece classification random cropping is not going to help as chess pieces, especially in the dataset used throughout this project, are always the same size relative to the sqaure. In contrast, both horizontal and vertical flipping do apply as pieces can be placed in any orentation within a square. In order to approach this selection of augmentation as systematically as possible, no assumptions are made and the hypothesis just made about which augmentations may relavent are tested empirically. See Figure 2.1 for an idea as to how augmentations were tested, although note that only a few augmentations are shown in that figure and is not an exhaustive list.

2.2.4 The Final Model

The final model contained two base models based on the ConvNeXT architecture. One of these specalised in occupancy detection and was trained with 'occupied' labels. The other was trained on the 'pieces' labeller with two heads, one for piece type classification and the other for piece color detection. The output from these two heads were concatenated and the CrossEntropyLoss is used against the 'pieces' labeller and optimised with AdamW.

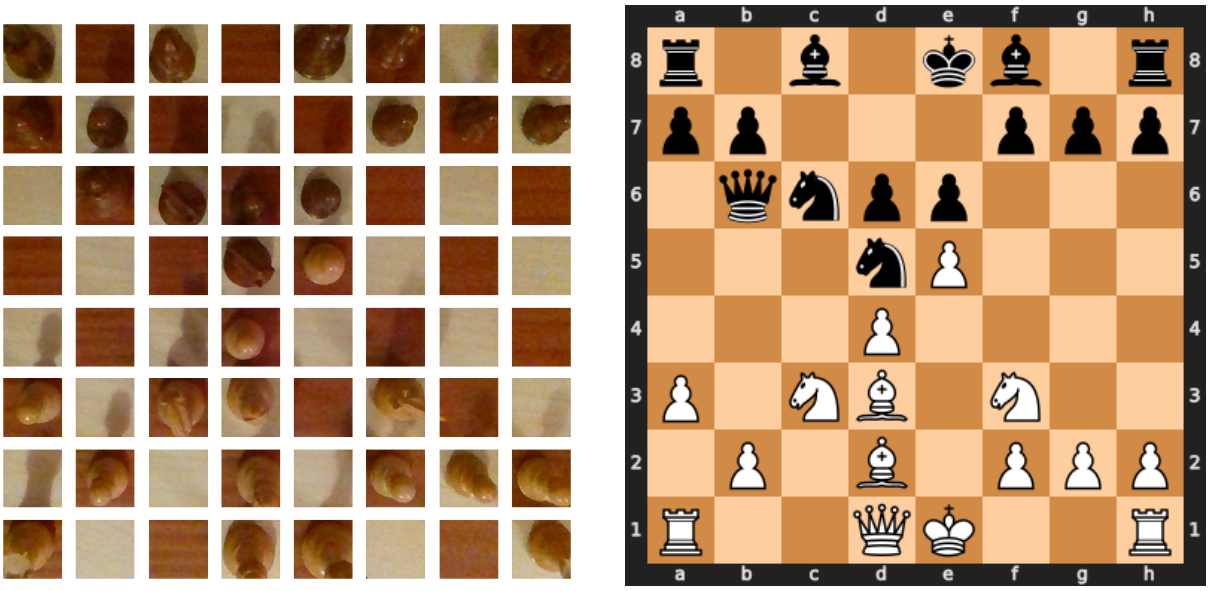
2.3 Recording a Chess Game to PGN

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 refererred 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.



(a) Frame after Segmentation

(b) Prediction Vision State

Figure 2.7: Board Squares Segmentation Process

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.

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 provide a better view of its utility.

3.1 Piece Recognition

The best openly available solutions for chess piece recognition on real boards were found to be from NAME [] and NAME []. There was the concern that through each iteration and optimization of hyper-parameters the presented solution was being overfit to the images in the evaluation set, despite never being directly trained on it. This could give the proposed solution an unfair advantage when testing on the original board. And so, in order to make a fairer comparison, another board entirely was also chosen for these final tests. Including a dataset from an unseen board should allow for more meaningful conclusions to be drawn regarding the generality of each model. Three tests, with the results shown in Figure 3.1 were then performed. The same training and evaluation splits were used for each model and the metrics were averaged over 3 runs. The first test used data from the original board only, the second with data from only the unseen board and the third test included data from both boards.

Two metrics are presented in 3.1: Accuracy and Balanced Accuracy. Accuracy was chosen due to its human interpretablility and intuitive definition as the percentage of correct classifications. Balanced Accuracy is equivalent to the arithmetic mean of recalls for each class which gives us some information that accuracy alone does not. When the test dataset is unbalanced, accuracy gives a view of how well each model performs over the whole dataset, but balanced accuracy gives the same weight to each class, therefore makes it easier to see problems with under-represented classes.

$$Accuracy = \frac{\sum_{i=1}^{k} tp_i}{\sum_{i=1}^{k} tp_i + fp_i} \tag{3.1}$$

$$Balanced\ Accuracy = \frac{\sum_{i=1}^{k} \frac{tp_i + tn_i}{tp_i + fn_i + fp_i + tn_i}}{k} \tag{3.2}$$

[] Uses a pretrained VGG16, freezing all layers of the convolutional feature extractor, adding a head of 3 fully connected layers with a sum of 2.5M trainable parameters. The final layer outputs a softmax distribution over all 13 class ('all' from Figure 2.6). After getting familiar with the code base and running some different experiments, 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.

	Original Chessboard		Unseen Chessboard		Both Chessboards	
Method	Accuracy	Balanced	Accuracy	Balanced	Accuracy	Balanced
proposed	**0.97**	**0.95**	**0.96**	**0.96**	todo	todo
[]	0.87	0.71	0.86	0.75	todo	todo
[]	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

[] Trains a Support Vector Machine classifier for each piece type including empty ('type+' from Figure 2.6) using the HOG features extracted from each training image. For each input image the classifier with the highest confidence determines the piece type, to determine piece color they use reference colors. This involved manually calculating reference colors for each piece and deciding on a suitable threshold value which was done separately for each board. Unfortunately, this method made it difficult to generalize when using multiple boards and so the decision was made to exclude it from the final test.

As can be seen the methods that employ CNNs dramatically outperform the more classical method using HOG descriptors which should not be surprising. The proposed method then, not only achieves 10% higher accuracy than the other CNN method, but it's balanced accuracy is almost the same it's accuracy, which cannot be said for the others. The main reason for this is the use of multitasking. The most common class in the dataset is the empty square and so when [] model is learning, the empty square gets more influence on the gradient of the weights to be updated. In turn this will end up with a model that has a high accuracy on the evaluation set, but the recall of under-represented classes will suffer as a result. One method to avoid this is to prune the dataset of empty squares, however correctly detecting empty squares is very important and the more that can be learnt from the data, the better. And so the proposed solution of using a separate classifier for this class has proved to be very successful.

3.1.1 Trials

For a clearer comparison in the holistic task of full chessboard state recognition a similar methodology to that of Ding, Czyzeqskil et al. [14, 9] is used. It involves directly comparing the predictions of each model on a small benchmark of chessboard images. This is in contrast to using the dataset used above which consists only of individual pieces and their corresponding labels.

Figure 3.3 truly demonstrates the extent of the challenge by visualizing the actual predictions from each model.

Interestingly [] method, while under-performing in accuracy on the evaluation tests, has a much higher recall of empty classifications compared to [] who utilizes a CNN. The reason for this also explains the gap in accuracy and balanced accuracy of the solution from Figure 3.1 and is that they used a separate classifier for each class. While the this works really well for the empty class, the HOG method does not perform as well on the other classes which have a more complex structure and it comes at a big computational cost as can be seen in Figure 3.2.

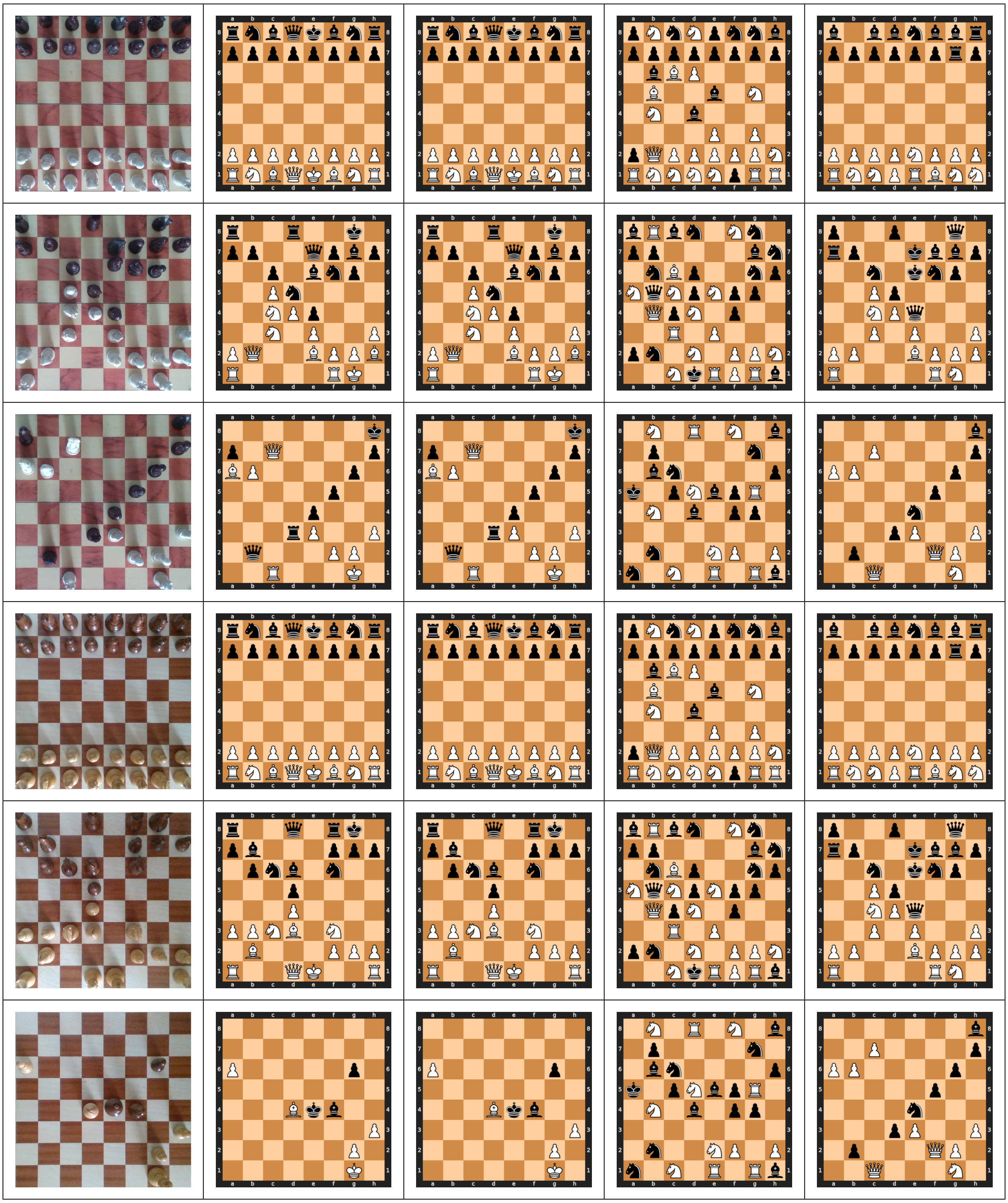
The speed of each model was also measured and averaged over 7 runs with the sample standard

deviation taken. A run in this context refers to passing 64 images (every square of the board) through the model and the value recorded in Figure 3.2 is the time for all of these images to be processed. The CPU was used for each model just comparison as [] method was not easily accelerated with a GPU, in the next section you'll see the actual performance of the proposed model is a lot faster with GPU acceleration as single instruction, multiple data (SIMD) is massively relevant with neural networks where the same operation such as convolution operations are performed many times over many input images which in itself can be tessellated and processed in parallel on the GPU.

Method	Mean Inference Time	Total Misclassified Squares	Cross Entropy Loss
Proposed	1.7s ± 61.4ms	-	3.01
[]	**1.5s ± 58.7ms**	-	3.21
[]	15s ± 417ms	-	1.82

Figure 3.2: Games played with the inference application

Input	Ground Truth	Proposed	[]	[]



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

Figure 3.3: A sample of six segmented chessboard images and their corresponding state prediction from each model

3.2 Recording PGN

In addition to evaluating the raw model's performance, five games of varying length are played from beginning to end with the inference application. The games were played as normal with the only exception being that moves were only made after the inference application detected the previous move. These delays, as they'll be refereed to in Figure 3.4, were measured with a timer and recorded. If the model failed to correctly generate the correct PGN at the end of the game, or otherwise crashed, it was marked as a failure.

			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 3.4: Games played with the inference application

The one failed game (Carlsen/Nakamura) was due to the inference application registering a rook movement before the player lifted their hand from the rook who 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.

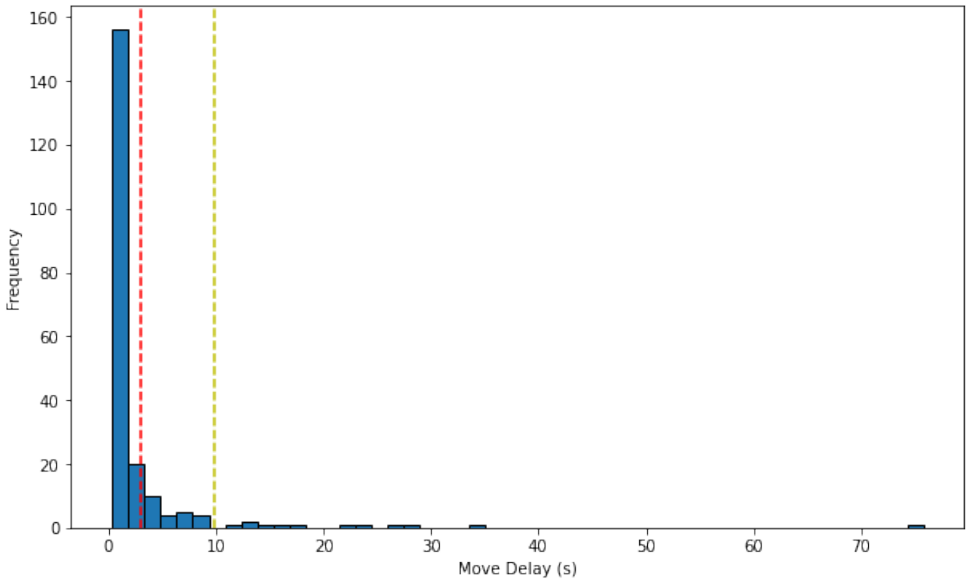


Figure 3.5: Histogram of move delay across all games in Figure 3.4 with arithmetic mean in red and sample standard deviation in yellow

While the delays on the tail end of the distribution are certainly caused by uncertainty in the model, the cause of the median delay is more difficult to pinpoint and so a profiler was used quite a lot through out the project and is shown here in Figure 3.6. A reasonable chunk of time is consumed by the display function which provides the user feedback as can be seen in the

provided demo. As this is not vital to the function of the application it would be possible to place this logic within another thread with a lower priority. This should completely remove ($\tilde{2}0\%$ of the total delay) if the machine has a free CPU to schedule the thread to. Due to using a GPU and inference the entire time spent forward propagating the images through the model took less than 5% of the whole inference loop, the most glaring time during inference seems to be spent in the `.item()` method of pytorch which is moving data from the GPU to the CPU.

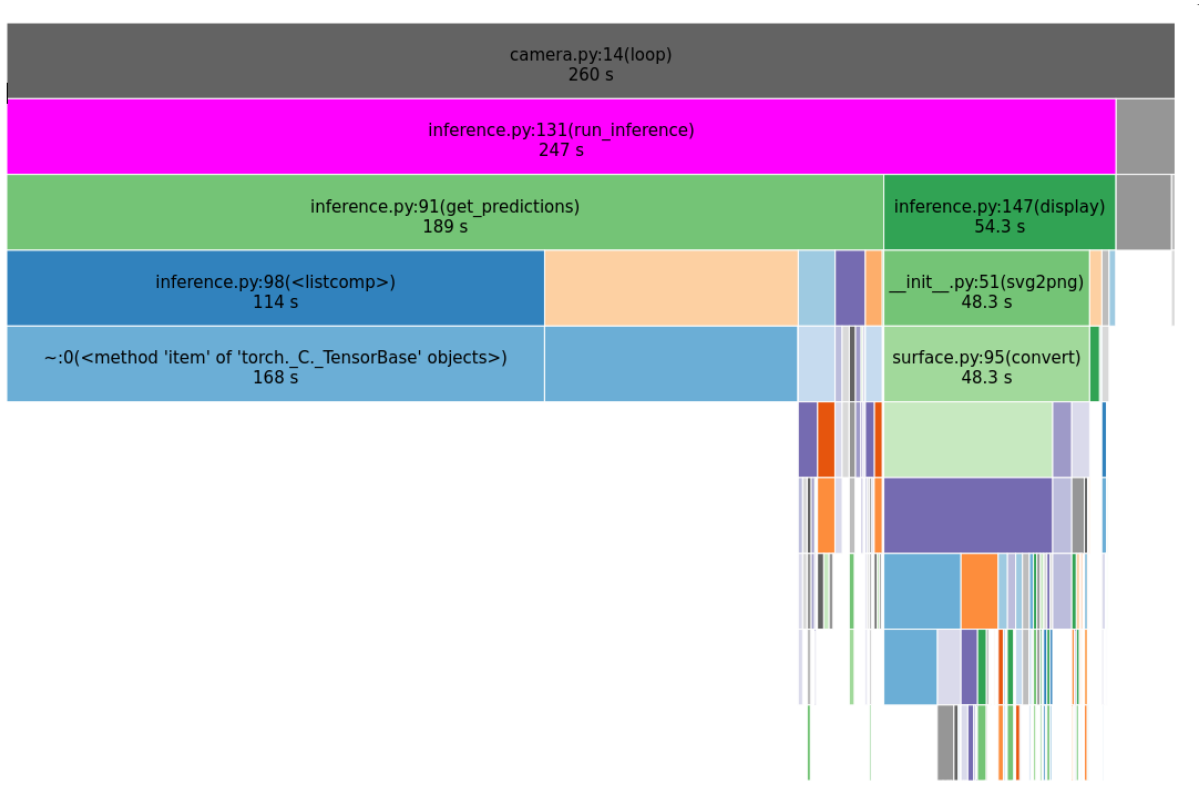


Figure 3.6: cProfile Visualisation of recording a 4 minute game at inference with moves made in quick succession

Chapter 4

Discussion

4.1 Conclusions

4.1.1 Data Collection

As expressed in section 2.1 building and managing the datasets to solve a particular problem is by far the biggest task. A creative and easy way to collect a large amount of chess data, label it and fit it to pre-existing network architectures has been found. And the tools to do so were made such that anyone could collect chess datasets with little effort. This was one of the main goals for the project and as promised the dataset as well as the tools I made to create it will be made open source in the hopes that the community can build a more diverse dataset of board from across the world. This will help in comparing models and training them to be used in a genuinely useful business setting.

It was not until I recorded and labeled lots of data (1000s images) that I suddenly started to see patterns in the data that were not visible before. Small things like motion blur and pieces half across squares. Only two boards were ever used throughout this project and so no doubt are more things to be found in larger datasets.

4.1.2 Board Segmentation

As this project focused on piece recognition and the development of an inference system, aruco markers proved to be a sensible choice for chessboard segmentation. They're really quick to detect, accurate in their localization and work even with pieces on the board which is very useful for data collection and inference alike. This is unlike a lot of other proposed solutions [].

However, they come with deal breaking consequences for any method that is to be used in a production setting with many boards. It's simply 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.

I would highly recommend the solution presented by [] to be explored in future work. While the method they propose can take up to around 5 seconds to segment the board they showed it to work in a huge array of environments and would be really interesting to see how it could be combined with the piece recognition system proposed in this report.

4.1.3 Piece Recognition

The Final Model presented for piece recognition outperformed the best open source alternatives by a considerable margin. When tested on an unseen, and rather difficult chessboard, it achieved 96% accuracy in comparison to the next best CNN model which achieved 86%.

The primary insights that contributed to this difference in performance is the exploitation of multitask learning with the state-of-the-art CNN architecture, ConvNeXT. Selecting which parameters to share and freeze during training and between tasks took a lot of experimentation but payed off. A good experiment tracking system that worked across machines was vital to be able to manage the some 2000 total models trained.

What was expected is that sharing weights between the 'type' classifier and 'occupied' classifier would aid in identifying empty squares since the features that make up an empty square are just the lack of features that make any of the classes in the 'type' classifier. It turns out, however, that the empty classifier trains more effectively when it has its own weights it can adjust as the proposed model shares very few weights between the 'occupied' classifier and every other class and the empty classifier in [] is completely independent as opposed to in [] where all the features are shared.

4.1.4 Inference

For most games the inference application developed for this report successfully exported 100% accurate PGN. It was a delight to use and very impressive to watch in person. Integration with a chess engine, implementing motion detection and introducing the concept of memory are the three additional components that can be accredited for the solutions reliability. The links to five sample recordings are included in Appendix C.

4.2 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 shift. This should be a fairly simple extension candidate if someone was looking to get involved in the codebase as most of components are ready to go. From here, it is very perceivable that a mobile app could be developed that utilizes this model. With a mobile app almost anyone could record their chess games without the need for a specialized camera setup. T he app could also integrate with a chess engine like Stockfish [] or AlphaZero to provide analysis later or even suggest moves. One consideration would be performance here as mobiles are more constrained by hardware than laptops and computers due to their size. It would be an interesting task to take advantage of lots of moderns phones deep learning hardware acceleration like Google's Tensor chip [], Apple's neural engine [] and Qualcomm's AI Engine [] and measure the performance.

If this solution is to enable autonomous robots in the game of chess then localising pieces in 3 dimensional space is a must have requirement. For this, it's 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. Future work could begin in 2D but instead perform instance segmentation across the whole input image, this may be able to provide enough information for a control system to use for manipulation planning. However, a more robust solution might instead want to perform 3D reconstruction from the

input image. Then object classification could be made on a point cloud. Point clouds are commonly used within robotic grasping solutions [] and so this would be the ideal space representation to work in.

References

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

[2] H. Bay, T. Tuytelaars, and L. V. Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

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

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

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

[6] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

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

[8] G. Cohen, S. Afshar, J. Tapson, and A. Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

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

[10] M. W. Davidson and M. Abramowitz. Molecular expressions microscopy primer: Digital image processing-difference of gaussians edge enhancement algorithm. *Olympus America Inc., and Florida State University*, 2006.

[11] 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.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

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

[15] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

[16] O. Eluyode and D. T. Akomolafe. Comparative study of biological and artificial neural networks. *European Journal of Applied Engineering and Scientific Research*, 2(1):36–46, 2013.

[17] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

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

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

[20] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[21] S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[22] L. Jianzhuang, L. Wenqing, and T. Yupeng. Automatic thresholding of gray-level pictures using two-dimension otsu method. In *China., 1991 International Conference on Circuits and Systems*, pages 325–327 vol.1, 1991.

[23] J. Kittler. On the accuracy of the sobel edge detector. *Image and Vision Computing*, 1(1):37–42, 1983.

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

[25] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[26] S. K. Kumar. On weight initialization in deep neural networks. *arXiv preprint arXiv:1704.08863*, 2017.

[27] S. Lazebnik, C. Schmid, and J. Ponce. Semi-local affine parts for object recognition. In *British Machine Vision Conference (BMVC'04)*, pages 779–788. The British Machine Vision Association (BMVA), 2004.

[28] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[29] T. Lindeberg. Image matching using generalized scale-space interest points. *Journal of mathematical Imaging and Vision*, 52(1):3–36, 2015.

[30] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. A convnet for the 2020s. *arXiv preprint arXiv:2201.03545*, 2022.

[31] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*, volume 2, pages 1150–1157. Ieee, 1999.

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

[33] J. Mukhoti, P. Stenetorp, and Y. Gal. On the importance of strong baselines in bayesian deep learning. *arXiv preprint arXiv:1811.09385*, 2018.

[34] F. Murtagh. Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5-6):183–197, 1991.

[35] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[36] L. Roberts. *Machine Perception of Three-Dimensional Solids*. 01 1963.

[37] 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.

[38] 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.

[39] L. N. Smith and N. Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial intelligence and machine learning for multi-domain operations applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

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

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

[42] A. Underwood. Board game image recognition using neural networks, Oct 2020.

[43] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[44] A. M. Wink and J. B. Roerdink. Denoising functional mr images: a comparison of wavelet denoising and gaussian smoothing. *IEEE transactions on medical imaging*, 23(3):374–387, 2004.

[45] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.

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.1.1 Positives

By far the strongest aspect of my report is the actual solution I produced. It performs better than any other openly available solution. The code itself has received a great deal of attention and utilizes many software engineering principals to ensure the solution is easy to read, extendable and performant:

- Environment variables are used to inject configuration into configuration classes
- Python generators were used to avoid loading many GB of data into memory
- Facades were used to abstract underlying implementations for Storage and Camera interfaces
- Proper experiment tracking was used to manage model versions and run experiments in the cloud with A6000 GPUs
- Debuggers and profilers were used to benchmark performance and find bugs

The results also demonstrated a rigorous and scientific approach to evaluating my solution, improving upon existing solutions and ensuring there was no bias when performing these tests. Yet I was also clear about where my solution had limitations and expanded on what could be done to mitigate those.

A.1.2 Negatives

I believe the report could be written better. Despite some effective use of figures and a decent structure, the language is not up to par with a lot of the published works I have read. There were a lot of things I felt I wasn't able to convey clearly or at all, however, this is a skill I have only just begun to stretch and I have found it enjoyable to exercise.

A.2 Personal reflection and lessons learned

Firstly found it to be a really fun project and enjoyed writing all the components that make a machine learning solution. Seeing the end result was even better as I'm genuienly surprised at how well it worked, beating all the open source solutions I could find out there. Learnt a lot about statistics which I never covered at A-Level.

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

A.3.1 Legal issues

Legal issues could only really enter my project through the data collection and usage aspects. By using my own data of games I played with no personal information I was able to completely avoid any legal issues. As I am opening my data and solution to be opensource as I found that I really appreciated when others did so legal issues may also enter such as licensing. Any datasets collected with my tool will be the complete responsibility of the person who captured that information and would need to be included in the license.

A.3.2 Social issues

The one social issue I can really see with this project is if it is used for cheating. I also believe it would be trivial to train my solution on digital games such as those on chess.com as the data is abundant and would probably work very well. This could lead to some bad actors using it to run analysis while playing the game and so would ruin the game for those who want to play against real humans.

A.3.3 Ethical issues

Personally, there weren't any serious ethical issues except those that I have already mentioned. The real ethics come into when the tools and techniques I have used throughout this project are used for other purposes. Deep learning is improving at a rapid rate and because of it's nature it's applicability knows really no bounds. I believe these techniques can be used for immense good and therefore should be avoided, however, the negative impact it could have if not careful could disastrous and something I am very passionate about keeping a very close eye on.

A.3.4 Professional issues

If this solution were to be used in a professional setting, such as recording chess games in competitions, and mistakes were made by the model, then I could see some pretty bad issues. I can't image Carlsen being very happy if he wanted to analyse his game but the inference application failed to record to PGN 100% correctly. This was not the aim of the project and more measures and effort would need to be taken to avoid consequences in professional settings.

Appendix B

External Material

B.1 Software Libraries

- pytorch (https://pytorch.org/)
- GuildAI (https://guild.ai/)
- python-chess (https://github.com/niklasf/python-chess)
- matplotlib (https://matplotlib.org/)
- numpy (https://numpy.org/)
- opencv (https://opencv.org/)
- pandas (https://pandas.pydata.org/)
- seaborn (https://seaborn.pydata.org/)

B.2 PGN File

Any external PGN files that were not automatically generated from my solution came from PGN Mentor (https://www.pgnmentor.com/)

Appendix C

Demos