Data Analytics MSc Dissertation MTHM038, 2021/22

Mockfish: A Pattern Recognition Chess Engine

Behavioural cloning of chess logic using convolutional neural networks.

Joshua Michael Griffiths

ID 210711042

Supervisor: Dr. Fatemah Parsa

A thesis presented for the degree of Master of Science in Data Analytics

School of Mathematical Sciences
Queen Mary University of London

Declaration of original work

This declaration is made on October 13, 2022.

Student's Declaration: I, Joshua Michael Griffiths hereby declare that the work in this thesis is my original work. I have not copied from any other students' work, work of mine submitted elsewhere, or from any other sources except where due reference or acknowledgement is made explicitly in the text, nor has any part been written for me by another person.

Referenced text has been flagged by:

1. Using italic fonts, **and**
2. using quotation marks "...", **and**
3. explicitly mentioning the source in the text.

Abstract

Chess is an extremely complex game that requires deep forward thinking and evaluation of future positions in order to master. However, when time restrictions come into play, the strongest players must utilise rapid pattern recognition and a "gut feeling" that comes with years of practice.

With an approach building on the work of Oshri et.al. [9], I present Mockfish, a pattern recognition chess engine built via behavioural cloning of human players using only the board state as input.

By using a dataset of 45,000 games comprised of 1,700,000 moves, I am able to train a series of convolutional neural networks to analyse board positions and present a probability distribution of favourable moves, using no search or evaluation functions.

Despite not using these traditional chess engine bases, Mockfish has demonstrated ability to beat Stockfish 15 up to skill level 11/20 at a search depth of 6, when under equal time constraints.

The codebase is available at `https://github.com/jerseyjosh/MockFish`.

Contents

Chapter 1

Introduction

1.1 Background

Convolutional neural networks have been one of the most important breakthroughs of computer vision and deep learning since the emergence of GPU accelerated learning algorithms in the mid-2000s. AlexNet was the first CNN to win the ImageNet classification benchmark, and its popularity led on to most architectures of the 2014 ImageNet Large Scale Visual Recognition Challenge using a CNN based architecture, the winner of which used over 30 layers [11].

The motivation behind convolutional neural networks, much like traditional neural networks, uses biological intuition. The interactivity of neurons is an attempted replication of animal visual cortex activity, where individual neurons respond only to stimuli in a restricted part of the visual field [5] - the kernel of the CNN layer. By projecting visual channels onto higher dimensional feature maps and then downsampling with pooling layers, CNNs can limit memory impact whilst learning spacially invariant substructures of an image, leading to impressively high image classification accuracy.

Traditional approaches to playing games have involved reinforcement learning methods, where the problem is framed as a Markov decision process.

In this framework there is a set of total environmental states, S, accompanied by a set of total agent actions, A, and the task is to identify a policy that maximises the reward given by a series of actions.

Alternatively, games can be broken down to a classification problem, where CNNs can be utilised. In this framework, game states are viewed as images, and the task then is to classify these images by what the optimal response to them is, in a traditional machine learning approach. If implemented in a quick enough manner, this can also result in an autonomous agent that is able to react to environmental stimuli dynamically and in a way that at least attempts to maximise some reward.

1.2 Applications to Chess

Chess is a two-player zero-sum game with perfect information, meaning that both players have access to the same entire board state at any single time, and are both making decisions based on the same input.

This setup means that chess is theoretically solvable by brute force. If one starts with a board position one move away from checkmate, then the best move, trivially, is the checkmating move. By logically continuing this, the opponent's best immediately prior move would be one to prevent the checkmating attack, and one's own best move prior to that would be one preventing their defence, and so on.

By continuing this process of building backwards from checkmated positions, a web of board states can be defined as either certainly won or certainly lost, given perfect play. The problem with this implementation in chess is that there are more possible chess game states than there are atoms in the observable universe [12], and there is no feasible way to evaluate this divergent tree of possibilities.

Instead, one must define some continuous loss function evaluated at different board positions, somewhere between won and lost. Then a perfect

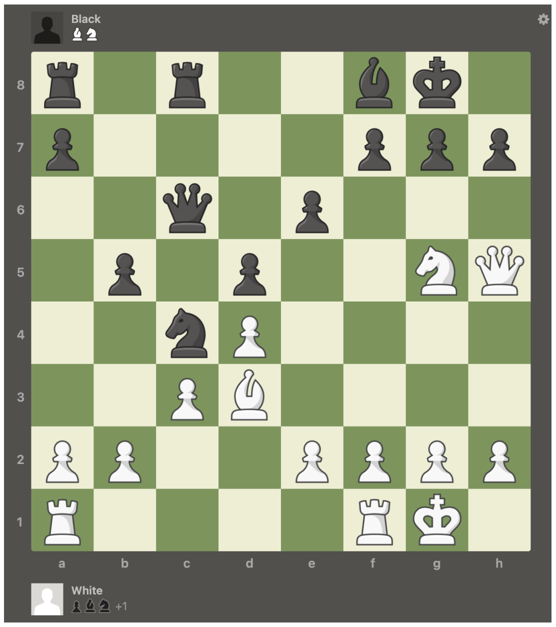


Figure 1.1: Example of a mating attack. Board generated from chess.com.

strategy would be to choose a move that minimizes the maximum possible loss from any given board position (the so-called minimax approach).

The most effective chess engines today are built using search and evaluation algorithms to prune this extraordinarily huge number of game states efficiently. Stockfish's search algorithm in particular uses alpha-beta tuning to reduce the number of nodes that need to be evaluated in the minimax process. This is an adversarial search algorithm that stops the evaluation of a node (candidate move) when another move is found that proves it is at least worse than a previously searched move. This ensures that game states that would not arise given perfect play from both sides are ignored.

In a basic example, Figure 1.1 demonstrates a position that, although not final, would be designated a won position for white, even with black to move next. If black were to play Pawn to h6, preventing Queen takes h6 checkmate, this allows Queen takes f7, King to h8, Queen to g6 and then inevitable checkmate by the Queen on h7. Even with perfect play from black, the board state is unrecoverable, and so would be considered with the same resultant score as a board with the black King actually in checkmate. Evaluation of this position can be safely ignored in the move search algorithm, so the search tree has been pruned. This reduces the computational depth of search required hugely in comparison to evaluating board states all the way to checkmate.

Evaluation functions, at their simplest, could be a point value count of each side's pieces, however the problem lies in capturing the dynamics between pieces and being able to quantify how strong one side's attack is against the other's defence. Stockfish's Efficiently Updateable Neural Network (NNUE) was introduced in its twelfth release, using a neural network to effectively evaluate the assigned score of non-final board states [8]. Since its introduction, all other top engines have used some variant of NNUE in order to remain competitive on the super-engine scene.

With this search and evaluation pipeline, there is no doubt that Stockfish's capabilities are far beyond superhuman in strength. However, as an engine it relies on prior chess domain knowledge:

- **Rules**

  Stockfish automatically limits candidate moves in its tree-search to legal moves given the board state.

- **Endgame Tablebase**

  Chess positions with 8 pieces or less remaining on the board have been exhaustively brute-force solved as of 2018[1], leading to easy perfect play for an engine with this prior information saved to memory. These endgame solutions are also used in the evaluation of non-final game states in order to avoid unnecessary searching.

- **Piece-Square Tables**

> In order to evaluate game states, Stockfish makes use of piece-square tables, or 8x8 grids of pre-calculated values for the positioning of each type of friendly and enemy piece. This prerequisite takes into account the relative value of each piece in its current position in relation to the game as a whole, and again helps to reduce the complexity of board evaluation.

Stockfish, and other engines, use this prior information to boost performance significantly, however the question is raised of whether there is a more biologically intuitive way of learning chess?

1.3 Behavioural Cloning

Extending imitation-style learning to autonomous agents has been much explored in literature, with the aim of being able to teach agents to react to environmental stimuli appropriately and quickly. Reinforcement learning has had much success in training AI systems to play games to a superhuman level, however the approach is not necessarily the most true to real biological systems.

As highlighted by Torabi et al. [14], many imitation learning paradigms fail to see that natural beings, when learning, do not understand the full action space in the same way autonomous agents are programmed to. Humans, when learning, operate entirely on sensory stimuli, and have no access to the actual mechanism of what they are trying to imitate.

Let us say a child is observing an adult stand up to reach for food from a table. The child sees the food as a reward and is encouraged to repeat the adult's behaviour, so tries to stand up as the adult did only to find themselves falling to the ground. In a reinforcement learning setting, the ability to stand up is a programmed element of the action space for the agent, and is executable on demand. The learning required is only that the child identifies walking in the adult, links it with a received reward, and then copies the behaviour themselves for similar reward.

In a true biological framework, however, the child does not know how to use their legs yet; that is part of the learning task in itself. If one is attempting to emulate this learning mechanism, it is important to be able to train agents to operate based only on sensory stimuli, with little other domain knowledge of the action and environment spaces that they inhabit. The child must be able to link the visual stimuli of seeing someone walk with the physical process of moving their legs in the same way, with similar results. This is a much more true representation of natural biological learning.

Humans, when learning chess, are not expected to perform in the way Stockfish and other engines do, by searching thousands of move combinations of perfect play and comparing all evaluated scores to find the optimal move. Much in the way a child must learn to stand from first principles, the human must also first learn the rules of the game, how the pieces move, and what checkmate means. This more natural style of learning comes often from watching better players at work, and attempting to identify these techniques in one's own play.

Chapter 2

Prior Research

2.1 Cloning Video Game Behaviours

Pierce and Zhu apply the idea of behavioural cloning to the game Counter-strike: Global Offensive [10], by treating the complex first person shooter as a series of classifications of game screen frames. They were able to take footage of humans playing the game and break it down into a series of training data pairs of images and user responses. By training a CNN on this data, they were able to create an autonomous agent capable of performing visually human-like game behaviours.

This contrasts importantly from the gameplay of a malicious “bot” account in the game, where hidden game data invisible to humans (enemy coordinates, gun recoil formulae etc.) is mined and used programmatically to shoot enemies easily. This style of AI gameplay is extremely easy to identify as the movement style is obviously non-human; the agent will aim at enemies through walls and miss no shots when firing their weapon.

An autonomous agent built using behavioural cloning principles does not display these giveaways. Pierce and Zhu note that their agent attempts to replicate the “recoil control” behaviours that humans must use when playing CS:GO, where one aims down in rhythm with the firing pattern of the gun

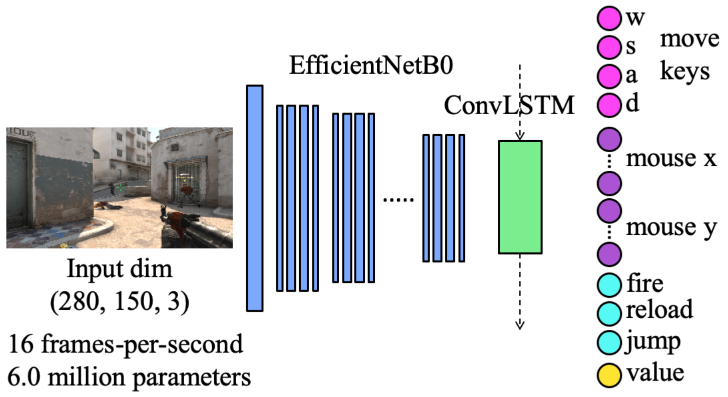


Figure 2.1: Behavioural cloning of Counterstrike: Global Offensive, from Pierce and Zhu [10]

they are firing. Without knowing enemy positions *a priori*, the agent is able to react to the visual stimulus of an enemy entering its field of view.

The challenges of CS:GO come from the fact it is run at a very high resolution compared to the traditional arcade-style games that game learning methods have been applied to prior, like Doom and the Atari arcade collection. This high resolution gives a huge memory cost in training deep neural networks, and requires downsampling of images in order to increase the network's ability to train.

Figure 1.2 shows the flow of an agent learning from an input image and classifying it to a selection of actions: moving, aiming (discretised), shooting, reloading, and other available actions.

Pierce and Zhu's results show that there is merit and utility in attempting a more intuitive learning style, in comparison to the brute force programmatic agents. By training using only visual stimuli, the learning process becomes much more in line with the way a real person would become good at the game, and perhaps this can be applied in other games.

2.2 Learning Chess

Good chess players are generally recognised as having the following skills:

- **Evaluation** The ability to see a board state and evaluate who is winning, based on positional motifs, relative piece value, time considerations etc.

- **Calculation** The ability to see future board states clearly after multiple threads of move sequences, and evaluate these positions effectively.

- **Pattern Recognition** The ability to immediately spot common tactics that arise in positions. Research has suggested that chess Grandmasters, the best in the world, remember up to 100,000 patterns [4] in their play. This skill has the indirect effect of pruning the search tree, as all it takes is recognition of a motif to recognise when a position is winning without having to mentally carry a position through to conclusion.

- **Intuition** The ability to internalise all of the above to an almost unconscious level, through years of practice and diligent training.

In classical, long format chess games, players spend hours evaluating positions and calculating far into the future to make decisions. In bullet and rapid formats, however, pattern recognition and intuition come much more into play. With only 60 seconds to play all of your moves in a chess game, there is almost no time at all to calculate variations far into the future, so one relies on their own instincts.

Evaluation and calculation are the most rigorous and quantifiable metrics of chess success, and are the focus of the biggest chess engines right now. The strongest engines in the world (Stockfish, Leela, AlphaZero etc.) all solely rely on the search and evaluate approach, a computationally expensive and

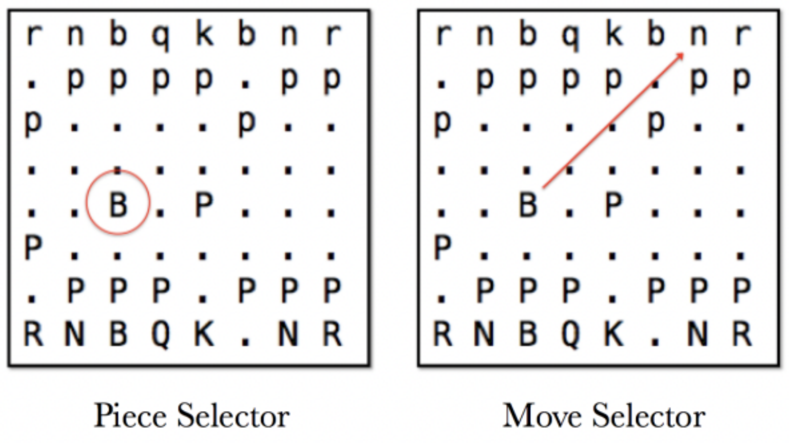


Figure 2.2: Model overview, from Oshri et al.[9]

resource intensive procedure. The question then arises; can pattern recognition and intuition in chess be learned behaviours, and could this improve the performance of engines?

Oshri et. al.[9] present an instance of using behavioural cloning to emulate a human style of play in chess, claiming that chess reasoning is a pattern recognition task that lends itself well to a CNN approach. In order to limit the cardinality of classes, their method involves training a network first to classify a square a piece should be leaving from, then training individual networks for each type of chess piece to classify where a piece should be moving to, resulting in 7 total CNN's that combine to predict moves.

This first **piece selector network**, they quote, *"capture(s) the notion of escape when a piece is under attack, or the king needs to move"*. The next **move selector networks** are networks trained similarly, but with one for each of the 6 types of chess piece. These networks take a board state and output selections for which squares each type of piece are most favourably placed on. I believe splitting the decision logic in this way does indeed bare resemblance to a human playing chess under time restriction, and is certainly far more human than a look-ahead tree search and evaluation approach.

Whilst they achieve successful results overall, their research is limited by computational constraints. Their training dataset consists of only 16,000 games, comprised of 196,000 moves. Whilst the exact architecture they use is not readily given, they quote that training on deeper networks proved unfruitful given the cost of grid-searching hyperparameters of increasingly large networks.

Despite these factors, their results are promising and they provide an engine that is claimed to be able to draw against a traditionally built chess engine, Sunfish [2], 26 out of 100 games, losing the rest. It is hoped that, with some optimisation, Mockfish is able to match these results.

Chapter 3

Methodology

3.1 Technical Setup

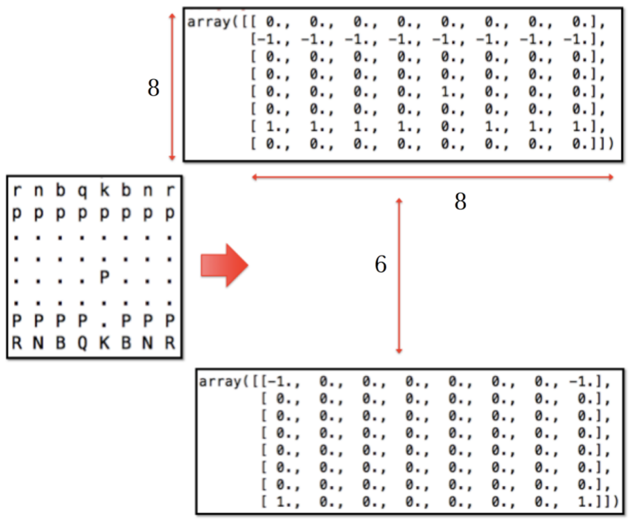
3.1.1 Data Structures

In the same style as Oshri et al. [9], I use a $6 \times 8 \times 8$ bitmap representation of a chess board, with a separate 8x8 layer for each type of piece. Separating piece types spacially in this way removes the need to specifically define piece values, as is the case in traditional chess engines. It is hoped that even without this prior knowledge, training on the data will give the network implicit encoding that, for example, a Queen is more important than a Pawn.

After experimentation with implementing piece-values, I found that there was no increase in accuracy, and in some cases even some decrease. This is understandable as it is generally accepted that CNNs perform most preferably with prior normalisation, so any layerwise multiplication of piece values will be removed by this step regardless.

3.1.2 Preprocessing

The chosen dataset is an exhaustive collection of all Computer-Computer and Computer-Human games played on the Free Internet Chess Server (FICS)



```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [-1., -1., -1., -1., -1., -1., -1., -1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.,  0.,  1.,  1.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . P . . .
. . . . . . . .
P P P P . P P P
R N B Q K B N R
```

```
array([[-1.,  0.,  0.,  0.,  0.,  0.,  0., -1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.,  0.,  0.,  1.]])
```

Figure 3.1: Array representation of board state, from Oshri et al. [9]

throughout 2016 [13]. Board states paired with the next moves made were generated by playing through each game in the dataset using the Python-chess package and flipping the board representation after each move, ensuring the networks remained agnostic to colour. This was also hoped to generalise tactical motifs more, as training from both sides of the board gives the engine experience of attacking both the left and right king sides. Further to this, the squares were labelled with numbers 0-63 that were flipped based on which side of the board one was viewing from, so as to ensure the square labels counted up from the bottom left to the top right, regardless of board orientation.

As the engine is being trained to make moves based solely on pattern recognition, it is not required (and is indeed infeasible) that any long term weaknesses induced by its moves are recognised. The most realistic expectation is that it does not learn to replicate one-move blunders that novice, or even intermediate, players may make.

For this reason, the dataset has been pruned to allow only games where both players have an ELO rating of over 2000. At this level it is hoped that the vast majority of moves are made following strong chess principles, and so moves should be of a high enough quality to be worth replicating.

Despite normalisation being the standard in neural network training, it had no effect on validation accuracy in initial experiments. I believe this is due to the nature of the data structure by comparison to traditional image recognition tasks. Three-channel RGB images for classification can have very different mean values, as dark images would all have pixels close to 0 in value and lighter images closer to 255. Normalisation in this case is beneficial as the training algorithm is hoped to learn more from structure and less from lighting. In the chess board bitmap representation, there is very small variation in mean between images, with the mean values being bounded s.t. $|\bar{X}| < 1$.

3.2 Training

3.2.1 Architecture

The baseline architecture of the neural network (Figure 3.2) is adapted from the Alexnet architecture [7], but with a 3×3 kernel and 1×1 padding in order to preserve the already limited resolution of the chess bitmap input. This choice of kernel keeps image size constant between convolutional layers.

After this, a variable length series of hidden layers is used with ReLU activations throughout.

3.2.2 Loss / Parameter Update

In training, I use a categorical cross-entropy loss in combination with an Adam optimiser. Whilst Oshri used a RMSProp optimiser, the added momentum component that Adam provides in combination with the heuristics of RMSProp is hoped to be beneficial for generalisation of models, and as such it has become the standard in much recent work.

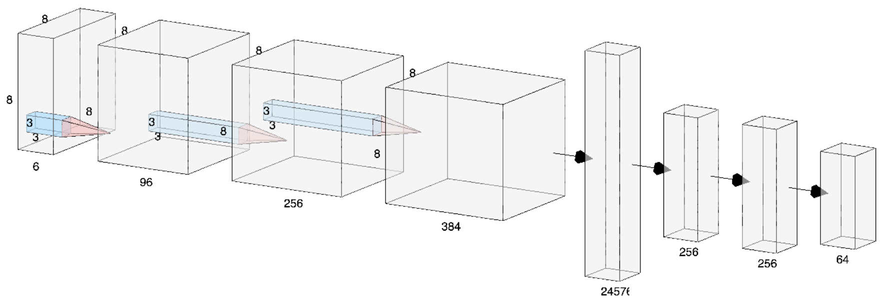


Figure 3.2: Baseline Model Architecture

3.2.3 Regularisation

Regularisation is traditionally used to improve the generalisation of learning algorithms, and to reduce an algorithm's tendency to simply memorise and replicate responses to training data. Whilst in some sense it is detrimental that a strong part of the engine's accuracy comes from simply remembering openings instead of focusing on more general tactical knowledge; I believe this is still in the spirit of the task of behavioural cloning and will contribute to a stronger and more human resultant playstyle.

As Mockfish is intended to be an engine that replicates the behaviours of real chess players, memorisation is a valid aspect of the learning process and is not something I believe should be purposefully avoided. Initial experimentation with L2 regularisation terms caused piece selection validation accuracy to hang around 46% indefinitely, so for the training of remaining networks it is ignored.

3.2.4 Software

By leveraging PyTorch MPS Acceleration, I am able to address computational issues in training on the larger dataset, some eight times in size compared to that of Oshri et al.

Training took place on an 8GB Apple M1 chip, using PyTorch 1.13.0, dev20220620. This developmental release, supporting Apple Metal Performance Shader acceleration, allowed greatly increased training and evaluation speeds by comparison to CPU. Hyperparameter tuning was performed using the Optuna package [3].

3.3 Validation

Due to the size of the dataset and limitations of training the networks on a single machine, cross-validation of training data is infeasible for tuning hyperparameters. Instead, 10% of the total dataset is left out for validation every 20% of each training epoch. Networks are trained until validation loss fails to increase 5 validation cycles in a row.

In the interest of keeping hyperparameter tuning time to a reasonable minimum, I choose not to attempt to tune the size and depth of the convolutional layers. As they are deep and wide-spanning compared to the resolution of the input images, it is hoped that any spacial features to be learned will have already been encoded through these, and the challenge remaining is finding a suitable architecture for the remaining fully connected layers to learn more complex relationships between the features.

Due to the time taken to train and evaluate each epoch of the network, and because I am allowing each training cycle to converge naturally, hyperparameter tuning is carried out through Bayesian optimization, using the Tree-Structured Parzen Estimator Algorithm. Whilst not exhaustive, this intelligent optimisation is hoped to find approximate maximisers for validation accuracy without having to grid search or random search samples of the hyperparameter space.

If convolutional layers remain constant between models then the following are to be tuned:

- **Number of hidden layers**

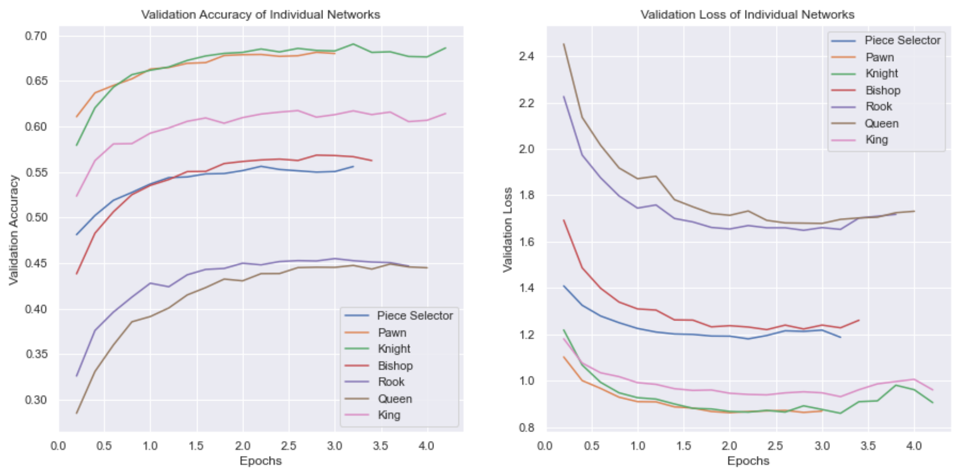


Figure 3.3: Training of all networks followed similar patterns, plateauing at approximately similar numbers of epochs.

The number of fully connected layers between the convolutional layers and the final classification layer, ranging from 1-5. Oshri et al. believed further increases in model performance would be achievable through inclusion of deeper fully connected layers.

- **Hidden Layer Size**

  The number of neurons in each fully connected layer. This is kept constant throughout multiple hidden layers (if applicable) to reduce further tuning, and is tested for values between 100 and 1000.

- **Dropout**

  Dropout layers added between each fully connected layer, with values between 0 and 0.7 with step size of 0.1.

- **Learning Rate**

  Although Adam is an adaptive optimiser, performance has been shown to increase when initial learning rate is tuned. This is carried out for values between 0.0001 and 0.0100 over a log-uniform distribution, as this helps in searching magnitudes of values.

- **Batch Size**

  To encourage GPU performance, training batch size was kept in powers of 2, and tested from 2^5 to 2^{10}.

With 20 iterations of the TPE algorithm, the most successful results are given by having 1 hidden layer of 561 nodes, a dropout rate of 0.2, learning rate of ≈ 0.000212, and a batch size of $2^7 = 128$. These options, when applied, increased accuracy across all networks when compared to performance of the baseline model: 2 hidden layers of size 4096, no dropout, batch size 256, learning rate of 0.001.

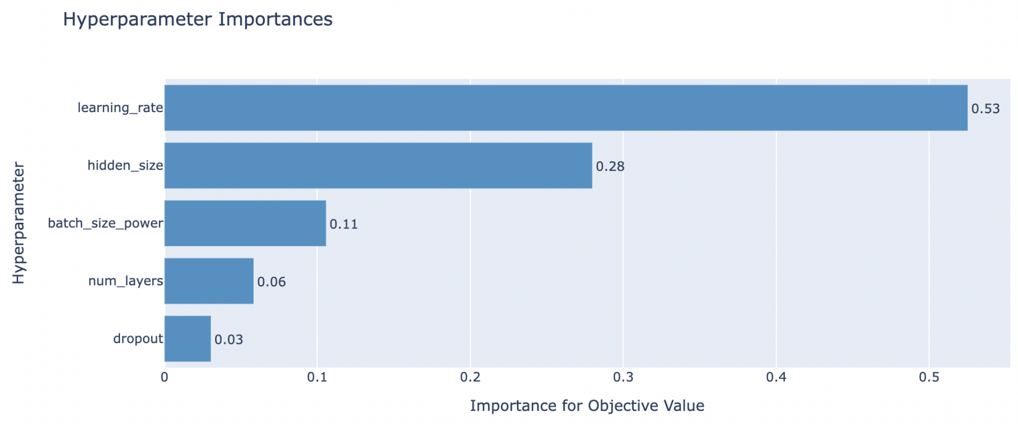


Figure 3.4: Hyperparameter Importance from Bayesian Optimisation, created with Optuna.

By repeatedly fitting random forest regressors with different hyperparameter values to predict the loss of a fully fitted neural network, hyperparameter importance can be estimated with a fANOVA analysis as outlined by Hutter, Hoos and Leyton-Brown [6]. The results of this are shown in figure 3.4.

Whilst learning rate and hidden size having the largest impact on convergence is unsurprising, it is interesting that dropout has such little effect.

Perhaps as we are basing training metrics on only the validation set, the effect of overfitting is minimised already.

Chapter 4

Results

4.1 Test Performance

The final models exceeded expectations across all networks, with results shown below. Here, classifications have not been clipped to only legal squares, so they are a reflection of the true 64-class classification task. Clipping of these to legal squares could only increase accuracy.

Network	**Test Accuracy**
Piece Selector	55.60%
Pawn	67.80%
Knight	68.84%
Bishop	56.72%
Rook	45.24%
Queen	44.80%
King	61.87%

As noted by Oshri et. al., accuracy for pieces with more local movements - Pawns, Knights, King - are the most accurate. This makes sense as it seems logical that if feature maps are taken from 3×3 localised kernels, then localised tactics may be replicated the best.

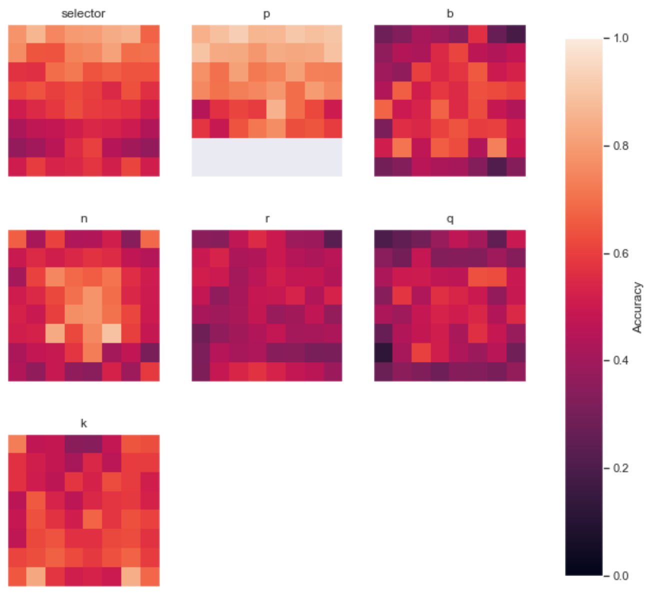


Figure 4.1: Confusion matrix of network accuracy across the chess board.

Despite the larger value pieces being less accurate, results here are also impressive, with the Queen move network being almost twice as accurate as Oshri's reported results.

It may be interesting to note also the accuracy of networks for each square of the board in order to detect any biases that they may face. Figure 4.1 shows these respective accuracies.

We can see here that the piece selector network seems reluctant to move pieces from the squares on the flanks of the second rank (a2, b2, g2, h2). The Pawns that sit on these squares at the start of a game are relatively uncommon to move early on, so this may reflect some level of opening theory that the piece selector network has learned.

It is also interesting to note that the piece selector chooses very accurately when pieces are on the eighth rank. This may demonstrate the fact that the engine has learned to immediately utilise Pawns that have just been promoted to more powerful pieces. This is reinforced by the fact that the pawn network seems to very accurately promote Pawns when it is able to do so.

The long distance piece networks appear to tend to prefer moving to the middle of the board rather than to the corners, which follows the chess principle of centralisation of powerful pieces, as that is where they exert the most pressure and control. The King network is also seen to most accurately castle when it is able to, demonstrating another learned principle of King safety.

4.2 Mockfish Engine

Testing accuracy is only a single metric for evaluation of the engine's learning, but I feel it does not truly reflect results. After all, chess is not a solved game, as it stands today, and there is no objectively correct move in every position. A truer reflection of the engine's ability is in analysis of behaviour in real chess games, in order to establish whether there seems to be some level of generalisation of tactical knowledge to unseen board states, and not just surface level mimicry.

4.2.1 Combining Networks

In order to interpret the results of the 7 neural networks as an engine capable of giving an ordered list of favourable moves, the scores for the networks must be combined in a meaningful way. This could be approached in different manners.

Composition

Initially, move selection was implemented by generating a 64×1 vector of scores for all 7 networks and then composing the piece selector network with all individual move selector networks, as in 4.1. Here, $\mathbf{s}_{e2}$ represents the piece selector score for leaving the e2 square, $\mathbf{m}_{e4}$ represents the move selector score for moving to the e4 square, and the product $\mathbf{s}_{e2}\mathbf{m}_{e4}$ represents the score for the move e2e4, for each $\mathbf{m} \in p, b, n, r, q, k$.

$$\begin{bmatrix} \mathbf{s}_{a1} \\ \mathbf{s}_{b1} \\ \mathbf{s}_{c1} \\ \vdots \\ \mathbf{s}_{g8} \\ \mathbf{s}_{h8} \end{bmatrix}_{64\times 1} \cdot \begin{bmatrix} \mathbf{m}_{a1} & \mathbf{m}_{b1} & \mathbf{m}_{c1} & \cdots & \mathbf{m}_{g8} & \mathbf{m}_{h8} \end{bmatrix}_{1\times 64} \tag{4.1}$$

$$= \begin{bmatrix} \mathbf{s}_{a1}\mathbf{m}_{a1} & \mathbf{s}_{a1}\mathbf{m}_{b1} & \cdots & \mathbf{s}_{a1}\mathbf{m}_{h8} \\ \mathbf{s}_{b1}\mathbf{m}_{a1} & \mathbf{s}_{b1}\mathbf{m}_{b1} & \cdots & \mathbf{s}_{b1}\mathbf{m}_{h8} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{s}_{h8}\mathbf{m}_{a1} & \mathbf{s}_{h8}\mathbf{m}_{b1} & \cdots & \mathbf{s}_{h8}\mathbf{m}_{h8} \end{bmatrix}_{64\times 64}$$

such that:

$$\mathbf{score}_{e2e4} = \sum_{\mathbf{m} \in pieces} \mathbf{s}_{e2}\mathbf{m}_{e4} \tag{4.2}$$

This approach assigned an objective score for all 64^2 possible moves (pairs of squares) on the board, which then gave an ordered list of the most favourable moves. By combining moves in this way, all networks are taken into consideration, and it was hoped that erroneous high scores for bad piece selection would be damped by their low respective move selection scores, or that the average over multiple networks would favour better moves.

Linear Combination

Moves could also be chosen in a more liner fashion. First the board state was input into the piece selector network, which outputted a score distribution across the 64 squares. The highest scoring of this output was then chosen as the square to move a piece from.

Then, based on whichever piece was in this square, the board state was fed into the respective move selector network to choose which square would be best to move to. The top scoring moves were then iterated down until a legal one was found, and this was chosen. If no legal move could be found, a backtracking-style algorithm was used to choose the next most favourable square to leave from and repeat the process from there.

Final Approach

The composition of networks and averaged vote quickly ran into issues, as move selection networks were voting for squares that their respective pieces could not even move to, and it became clear that it made no sense having this distributed responsibility. The engine made nonsensical moves and would in some cases simply remove the opponent's pieces from the board. Whilst this could be considered an innovative tactical decision, it was ineffective for the most part.

Linearly combining moves was also quickly found to have limitations, coming down to the fact that the piece selector and move selector networks were acting independently of each other. Pieces that were considered favourable to move may have very low scoring legal squares to move to, resulting in the engine making quite clearly bad moves in positions where there should have been obvious moves to make.

In order to find a middle ground, the eventual solution was to iterate through all squares of the board and multiply the piece selector score of that square by each move selector score of the network of the piece that sat in it. By clipping all empty squares and squares with enemy pieces in them to 0, and by clipping all illegal move squares to 0; we are left with a list of all legal moves and their associated scores. This results in a similar 64×64 matrix of possible moves, but where only relevant networks are contributing score, and piece selection score is also taken into account.

The resultant move matrix is given below.

$$\begin{bmatrix} \mathbf{s}_{a1}\mathbf{r}_{a1} = 0 & \mathbf{s}_{a1}\mathbf{r}_{b1} = 0 & \cdots & \cdots & \cdots & \mathbf{s}_{a1}\mathbf{r}_{h8} = 0 \\ \mathbf{s}_{b1}\mathbf{n}_{a1} = 0 & \mathbf{s}_{b1}\mathbf{n}_{b1} = 0 & \cdots & \cdots & \cdots & \mathbf{s}_{b1}\mathbf{n}_{h8} = 0 \\ \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \mathbf{s}_{e2}\mathbf{p}_{e4} > 0 & \cdots & \vdots \\ \vdots & \vdots & \cdots & \vdots & \ddots & \vdots \\ \mathbf{p}_{h8}\mathbf{r}_{a1} = 0 & \mathbf{p}_{h8}\mathbf{r}_{b1} = 0 & \cdots & \cdots & \cdots & \mathbf{p}_{h8}\mathbf{r}_{h8} = 0 \end{bmatrix}_{64\times 64} \tag{4.3}$$

To prevent the engine from being completely deterministic, there needed to be some level of random choice. This was implemented by applying a softmax to the pool of moves with positive combined move scores, in order to treat them as probabilities whilst discarding illegal moves.

This had notable increase in the experience of playing against the engine. In unclear opening positions where there are lots of possible moves, it chooses logical openings at random, but still favours a certain set of openings. However positions where there are only a few moves that prevent immediate loss of material or checkmate, the engine is more sure of its decisions, and tends to defend and attack in a relatively human manner.

4.2.2 Engine Playstyle

In my personal games against Mockfish, I have noted an especially strong grasp of opening theory. Repeated board states were not removed in the training data as it was felt that memorisation of popular openings was in the spirit of replicating a human style of play.

Further into the middle game, it becomes clear the engine plays quite defensively, and highly favours the trading of pieces. It is relatively good at saving pieces that are under attack, but if given the opportunity is quick to repeat moves, leading to drawish games if the other player is not aggressive.

Whilst Mockfish knows to attack high value pieces, it lacks the capability to capitalise on blunders and does not tend to see easy checkmating attacks. In hindsight, it is assumed this is because at the 2000 Elo level, many games do not result in checkmate, as players know when to resign losing positions. Perhaps if training data only included games played to the very conclusion, this issue could be amended.

4.2.3 Puzzle Training

To investigate the effect of training a more aggressive version of the engine, the entire puzzle database of Lichess.com was downloaded and a new engine was trained in the same way on these data, the testing results of which are below.

Elo here was not pruned as low Elo rated puzzles involved the easiest of checkmating tactics, something that could only help a more aggressive engine.

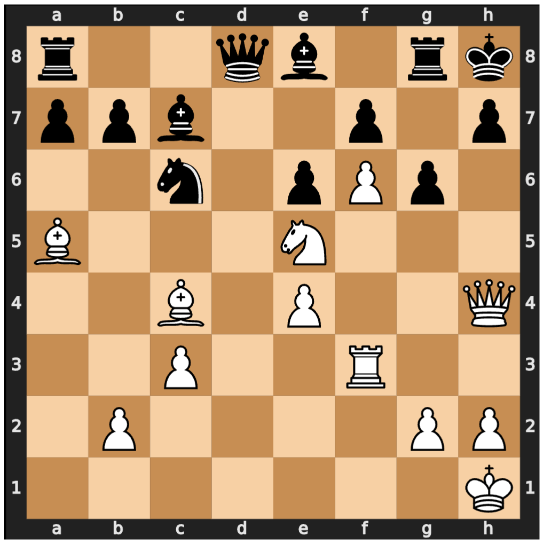


Figure 4.2: White has a checkmating attack.

Puzzle Network	**Test Accuracy**
Piece Selector	70.23%
Pawn	82.24%
Knight	80.17%
Bishop	76.93%
Rook	78.09%
Queen	73.97%
King	78.45%

Despite the astounding improvement in testing accuracy, it is quickly clear that this version of Mockfish is not genuinely playable. The engine is extremely confident in sacrificing pieces, which is often the approach in chess puzzles involving checkmating attacks, however it does so in opening positions for little benefit.

The difference of playstyle can be seen by comparing how both engines consider the chess position of 4.2. Here, the white side has an unstoppable checkmating attack by sacrificing the Queen on h7, forcing the King to recapture. This is then followed by Rook to h3 - checkmate.

When given to the standard implementation of Mockfish, the engine outputs scores given in 4.3, and chooses Bishop captures Bishop on c7 as the best move in the position. Even this is quite a substantial achievement as the engine has identified that the Bishop on a5 is undefended and wants to trade it off the board to solve this.

Furthermore, it identifies Queen to h6 as a favourable Queen move. This idea of having a Pawn on f6 and infiltrating h6 and then g7 with the Queen to deliver checkmate is a very common attacking motif. However, in this case it fails due to the Rook defending from g8.

Knight takes f7 is also seen as a favourable move, as it attacks the Queen whilst also putting the King in check, winning the enemy Queen on the next move if not for the fact the Knight would be captured by the Bishop on e8.

The engine also sees both Rook to d4, a defended square that attacks the enemy Queen, and Rook to h3, doubling up with the Queen to create a checkmate threat on h7 on the next move.

Although all impressively cognisant ideas, the forced checkmating line is ignored. If we consider now the scores outputted by the puzzle-trained engine in 4.4, there is a tangible difference in process.

The piece selector network is now much less confident of which piece to move, and most notably out of the move selection network, the engine now wants to sacrifice the Queen on h7 more than any other move, and indeed this is the move the puzzle engine makes, following up with checkmate on the next turn.

4.2.4 Against Stockfish

In order to evaluate performance in a fair manner, Mockfish was played against Stockfish 15 using multiple depth settings and skill settings, parameters which can be augmented to manipulate the strength of the engine. Overall, 500 games of each colour were played between the siblings at each

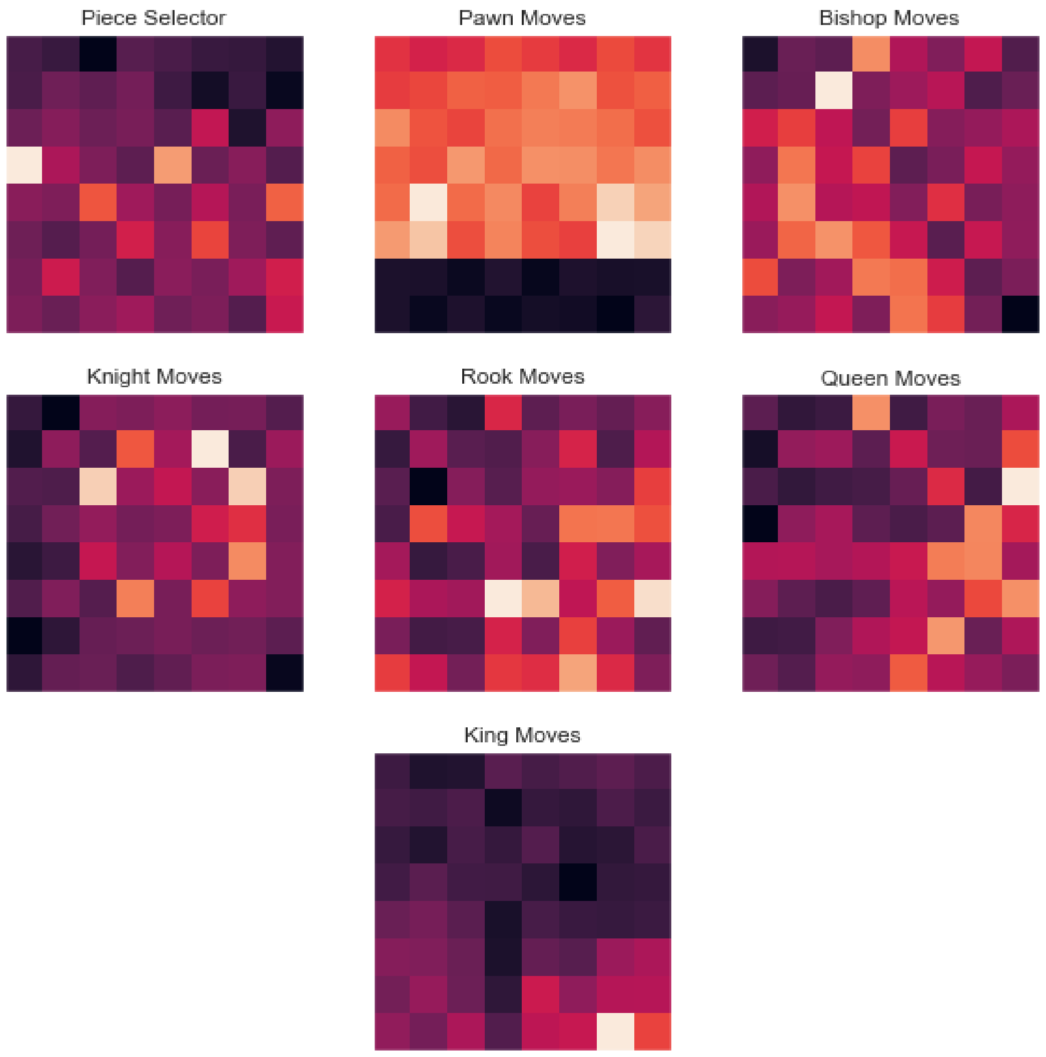


Figure 4.3: Mockfish standard view. Note how the engine favours the quiet Qh6, and is confident that the pieces on a5 and e5 must surely be moved.

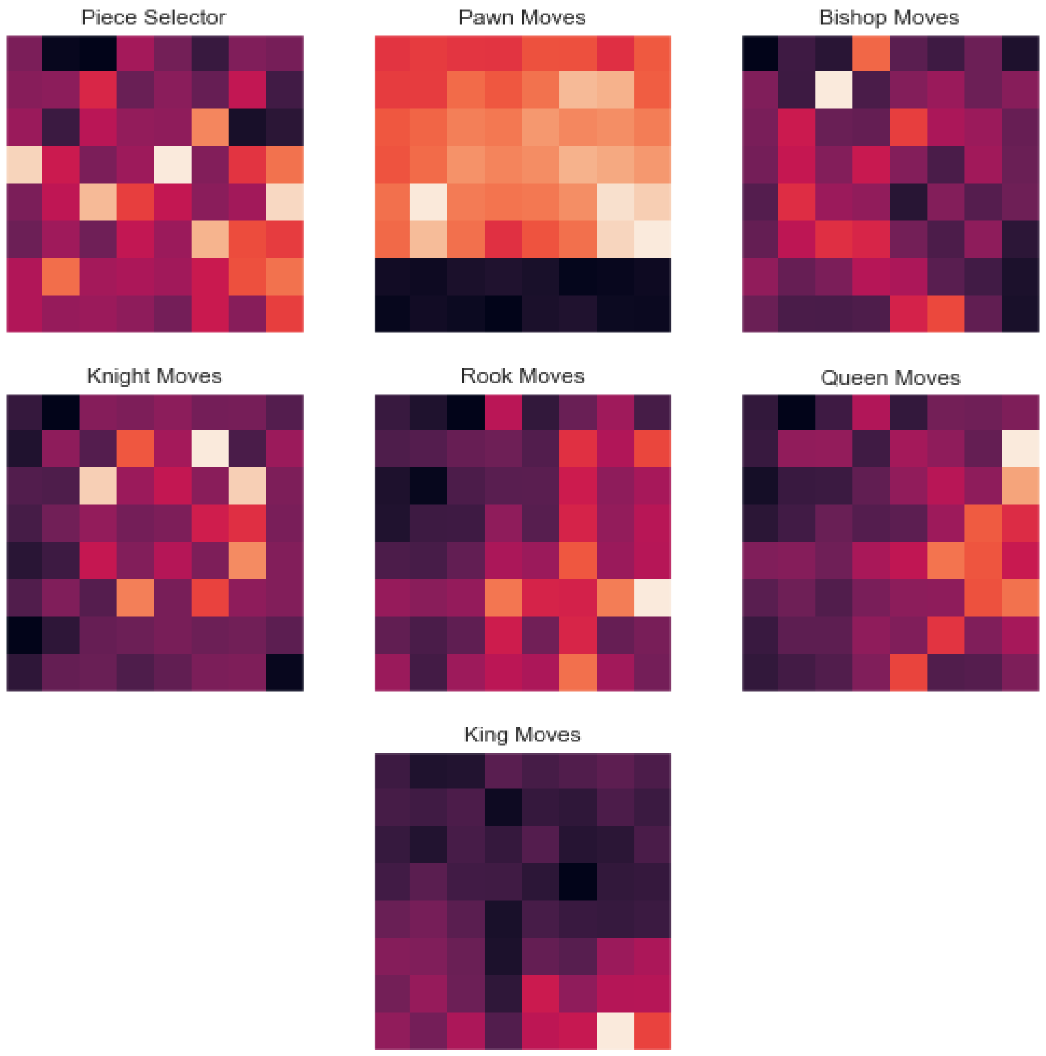


Figure 4.4: Mockfish puzzle view. Note how now the engine now favours the more aggressive Qxh7 checkmating attack, and is less sure of which piece is correct in the piece selector network.

skill level, taken from the playable Lichess.com Stockfish AI levels. The time Mockfish took to see the board and ouput a move was the time alloted to Stockfish to make its move.

Stockfish Level	**Win Rate**	**Draw Rate**	**Lose Rate**
1	2.1%	9.6%	88.3%
2	1.5%	5.1%	93.4%
3	0.9%	2.6%	96.5%
4	0.4%	1.1%	98.5%
5	0.0%	0.0%	100%
6	0.0%	0.0%	100%
7	0.0%	0.0%	100%
8	0.0%	0.0%	100%

In analysing drawn games, Mockfish withstands attacks and trades down to positions with insufficient material for either side to checkmate, or to positions where either moves are repeated or 50 moves occur with no captures or Pawn moves, triggering the 50-move draw rule.

The puzzle engine fairs much worse against Stockfish, and is yet to win a game against even the lowest level.

Chapter 5

Conclusion

Mockfish has an evident memory of opening theory, as can be felt by playing against it. Further to this, it is clear by the distribution of move selector network scores in middlegame positions that the engine has learned tactical motifs beyond just memorisation of positions. It frequently seeks to attack pieces, threaten checkmate and defends simple threats well, and it has clear knowledge of the legal moves a piece can make.

Where it fails, however, is being able to progress a game favourably and execute plans. The nature of intermediate chess encourages dealing with threats before they present themselves, so perhaps the training data, whilst encouraging good chess habits, is not sufficient for learning the logic behind them. Whilst the engine is capable of replicating seemingly human play, in any context outside of hyper-bullet time control, it can be easily outplayed.

Also, traditional engines like Stockfish are extremely computationally heavy and require hundreds of thousands of positions to be evaluated in order to make a decision, whereas Mockfish can infer a list of candidate moves from a board state almost instantly. Although alone it evidently poses no threat to traditional chess engines, there may be merit to pattern recognition software in combination with other search and evaluation methods.

In my opinion, the most promising aspect of the research relates more to the domain of behavioural cloning than mastering chess. As the approach outlined here picks up both opening theory preference and some tactical knowledge, further work may be able to get closer to creating engines that are tuned for players to train against. Strong engines have the drawback of being extremely inhuman to play against, and blunders must be coded in artificially to reduce strength.

An engine such as Mockfish may be trained to purposefully hang pieces, leave checkmating opportunities, or other learnable behaviours. Perhaps this may be more useful for amateurs to practice the game, as these behaviours can be tuned to what the user feels most beneficial, and may give a more authentic experience of playing against another person.

5.1 Future Research

If one wanted to shy away from opening theory and focus more on learning generalised chess tactics, regularisation may be something worth looking into. Although early experimentation led to plateaus in validation accuracy, validation loss continued to reduce, showing that weights were being smoothed whilst maintaining accuracy. This is indicative that the information required to predict moves was being embedded in some other way than memory and recall, and whilst it was generally unsuccessful here, further changes in methodology may improve this. The dataset, for example, could be edited so as to remove duplicate board states. This could further encourage learning to move away from memorisation.

There is surely more to gain from use of the puzzle-trained networks also, as the performance of the puzzle engine was far superior in terms of attacking principles to the original Mockfish, but failed in carrying out general good chess principles like piece development. This could maybe be implemented through some form of transfer learning, whereby the fully trained Mockfish weights are trained further with batches of puzzle training data and validation accuracy on both datasets are monitored to find a balance between the two. There is also the possibility of training a network to detect where tactics are available in a position, and then choosing which engine to push a move forward based on the outcome of that.

It is also, however, worth noting that there must be somewhere to draw a line on when further network introduction is necessary. With all standard Mockfish networks, all puzzle networks, and a puzzle identification network, the engine would come to 15 neural networks of approximately 60mb each, and with minimal improvements on performance, there are diminishing returns as complexity increases.

Overall, there may be some utility to be gained from rapid evaluation of board states using pattern recognition in conjunction with traditional search and evaluation algorithms. The engine's ability to hold up against the world's best chess engine has demonstrated this. However, the lack of logical continuation between moves does not make pattern recognition an effective approach on its own.

Bibliography

[1] Syzygy 7-piece tablebase. http://tablebase.lichess.ovh/tables/standard/7/. Accessed: 26/08/22.

[2] Thomas Ahle. Sunfish engine. https://github.com/thomasahle/sunfish. Accessed: 02/09/22.

[3] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[4] Ognjen Amidzic, Hartmut J Riehle, Thorsten Fehr, Christian Wienbruch, and Thomas Elbert. Pattern of focal γ-bursts in chess players. *Nature*, 412(6847):603–603, 2001.

[5] DH Hubel and TN Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of Physiology*, 195(1):215–243, 1968.

[6] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. An efficient approach for assessing hyperparameter importance. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 754–762, Bejing, China, 22–24 Jun 2014. PMLR.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[8] Yu Nasu. Efficiently updatable neural-network-based evaluation functions for computer shogi. *The 28th World Computer Shogi Championship Appeal Document*, 2018.

[9] Barak Oshri and Nishith Khandwala. Predicting moves in chess using convolutional neural networks. 2015.

[10] Tim Pearce and Jun Zhu. Counter-strike deathmatch with large-scale behavioural cloning. *arXiv preprint arXiv:2104.04258*, 2021.

[11] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[12] Claude E. Shannon. Programming a computer for playing chess. *Computer Chess Compendium*, 41(314), Mar 1950.

[13] @thule179. 2016 internet chess games. `https://data.world/thule179/2016-internet-chess-games`, 2017.

[14] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.