Playing Chess Endgames using Reinforcement Learning

Sunil Deolalikar[*]

In this paper we examine the strategy of building a chess engine that makes intelligent move decisions based on reinforcement learning. This engine is trained and applied on canonical chess "endgames" corresponding to simplified chess positions in which there are only a few pieces left (thus tend to occur near the end of a standard match). The engine has no strategic knowledge of chess positions and is trained by repeated self play of randomly seeded positions until game terminating positions are obtained.

I. Introduction

Strategies on building effective computer chess engines have been actively studied for years. Many current top chess engines make move decisions primarily by utilizing brute force searches, thus requiring great computational effort. Even with current computing power, completely analyzing all possible moves to high depth is infeasible, thus many move search pruning strategies are used. Due to the computational difficulty of evaluating chess positions it was only in 1997 when the most advanced computers were capable of defeating top human chess players with the engine "Deep Blue".[3]

Reinforcement learning techniques for building chess engines have been examined but are largely very difficult to make effective due to the absolutely massive state and action space that must be covered. After 10 moves there are estimated to be 169,518,829,100,544,000 board positions/states – clearly too many states to possibly explore.[2]

Therefore in this project we choose to look at a subset of chess problems called "endgames". These are canonical positions that tend to occur near the end of a match after many pieces have already been captured. The primary endgame examined here is a King and Rook vs King position which is commonly taught to beginners (see appendix for information on how these pieces move). These positions have guaranteed strategies for white to win, thus the metric used to test the ability of the computer after training is to check whether it is able to reach a checkmate from any given king and rook vs king position.

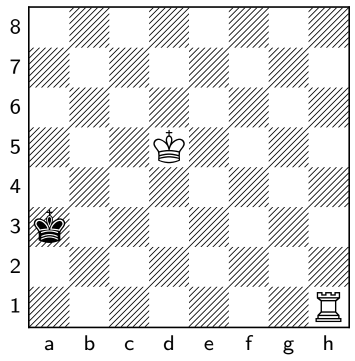

Figure 1. An example King and Rook vs King endgame

[*]Aeronautics and Astronautics Department, Stanford University

It is important to note that even with this simplified position we still need to apply efficient algorithms and exploit the problem structure to get reasonable results. The number of states are approximately on the order of magnitude of $P_{64,3} = \frac{64!}{(64-3)!} \approx 250000$ with about 10 to 20 distinct actions per state.

II. Problem Formulation

II.A. Game Representation

The board state representation is quite simple for this problem as we have a reduced chess position of just three pieces. The state is just stored as a dictionaries of the white pieces and black pieces. The actions are computed on the fly directly from the state (as the actions are often different depending the state). I use a standardized chess move generation procedure[4] where I first generate all moves based on the pieces present on the board and then filter these with game rules to return the set of only the "legal" moves from which an agent can make a decision.

The state time step is defined as the board state in which white is about to move. Therefore the transition between one state to the next is non-deterministic as the opposing player has a chance to pick their action between a timestep update. This intermediate step of the opposing player can be denoted as a "half-state" which we later abstract away automatically by just considering there to be a stochastic transition from $s_{t,white} \rightarrow s_{t+1,white}$ with $a_{t,black}$ being the noise factor.

We also have the reward function $R(s)$ as a pure function of state per time step. The function is chosen to returns a positive constant if the position is a win for white or a negative constant if the position is a draw for black given by the relation:

$$R(s_t) = 50\mathbb{1}(s_t = win) - 20\mathbb{1}(s_t = draw) \tag{1}$$

These dependencies can be explicitly drawn with a markov decision diagram [2].

$$\begin{tikzcd}
a_{t,white} \arrow[dr] & a_{t,black} \arrow[dr] & \\
s_{t,white} \arrow[u] \arrow[r] \arrow[d] & s_{t,black} \arrow[u] \arrow[r] & s_{t+1,white} \arrow[d] \\
r_t & & r_{t+1}
\end{tikzcd}$$

Figure 2. Markov Decision Diagram

For this project the decisions made by $a_{t,black}$ were based on a single greedy move look ahead. If the intermediate state $s_{t,black}$ provided an action $a_{t,black}$ that would instantaneously draw the position, black makes that move, otherwise a move is selected at random. This way the chess engine will be trained to not make one move blunders (in this case that is often moving the rook into a position where it can be captured resulting in an immediate draw) but will still be able to explore the state space well due to the usually random movement directions of the black king.

Finally we can take advantage of the board symmetry to split the chessboard states into quartiles. As the board is symmetric about the middle rows and columns, any given position has three other flipped positions which can all be accounted for in just state and action pair.

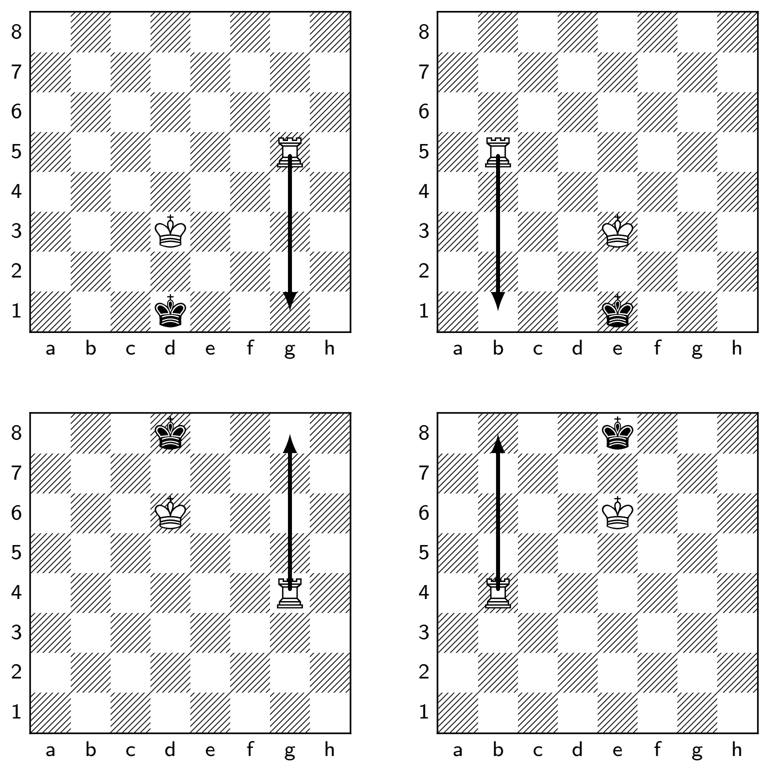

Figure 3. Examples of a set of symmetric positions and the symmetric optimal action

Thus to take advantage of this structure any position encountered by the chess engine is transferred into the coordinates in which the white king is in the bottom left quadrant (corresponding to top left diagram). Therefore even if we are playing in a different quadrant, the learning algorithm is converting those positions/moves into the left-bottom symmetric position and is only reading and writing to those values. This yields a factor of four decrease in the number of states letting the engine more effectively explore the state space.

Algorithm 1: Symmetric Game Position update

```
Given initial state $s_t$
while true do
    $s_{t,flipped} \leftarrow \text{FLIP}(s_t)$
    $a_{t,flipped} \leftarrow \text{CHOOSEMOVE}(s_{t,flipped})$
    $a_t \leftarrow \text{INVERSEFLIP}(a_t)$
    $s_{t,black} \leftarrow \text{UPDATEPOSITION}(s_t, a_t)$
    $a_{t,black} \leftarrow \text{CHOOSEMOVEBLACK}(s_{t,black})$
    $s_{t+1} \leftarrow \text{UPDATEPOSITION}(s_{t,black}, a_{t,black}$
    if $s_{t+1}$ = game end then
        break
    end
    $s_t \leftarrow s_{t+1}$
end
```

As we delegate move choosing and training to our agent within the chooseMove function, we have completely decoupled our agent from 75% of the possible board positions! Now it only needs to keep track and make decisions based on a greatly reduced set of flipped states.

II.B. Reinforcement Learning

For this paper we use the $Q(\lambda)$ learning algorithm [2]. This algorithm performs the standard Q state action value function update with:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a_t} Q(s_{t+1}, a) - Q(s_t, a_t) \right) \tag{2}$$

where α is the learning rate used to help let a given Q value converge and γ is the discount factor to emphasize actions leading to rewards sooner rather than later.

The λ refers to performing an eligibility trace or looping back through previously visited state and action history and distributing $\delta = (r_t + \gamma \max_{a_t} Q(s_{t+1}, a) - Q(s_t, a_t))$ to those states in decaying value in order of how recently they were visited. This is especially useful and important for problems such as this chess endgame as the rewards are very sparse, so when a state resulting in a win is reached, not only will the previous state value function increase, but also those of the sequence of state/action pairs leading up to this final state.

Algorithm 2: $Q(\lambda)$ learning algorithm

```
Given initial state $s_0$
Initialize $Q(s, a) = 0 \ \forall s, a$
Initialize stateHistory, actionHistory empty
while NUMGAMES < TOTALGAMES do
    $a_t \leftarrow$ CHOOSEMOVE($s_t, a_t$)
    Perform action and observe: $s_{t+1}$ , $r_t$
    $\delta = (r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$
    stateHistory.push($s_t$)
    actionHistory.pop($s_t$)
    iter = 0
    while |STATEHISTORY| > 0 do
        s = stateHistory.pop()
        a = actionHistory.pop()
        $Q(s, a)$ += $\alpha\delta\lambda^{iter}$
        iter++
    end
    numGames++
    if checkGameOver() then
        $s_t \leftarrow$ RANDOMSTATE()
        stateHistory.erase()
        actionHistory.erase()
    end
end
```

As we are playing a chess game here the model needs to account for an "end state" at which we cannot transition to a later s_{t+1} as the game has ended. This is handled by letting the update $\max_a Q(s_{t+1}, a) = r_t$ so that we do not need to compute the value of an post-terminal state. Additionally to keep training after a game ends, the state is reset using a random board seeding function and the process repeats until the desired number of training games are played through. Note that the state and action histories are also reset with the game ending condition as it doesn't make sense updating those values anymore after a randomized restart.

The move choosing algorithm is designed to create a balance between exploring the state space and optimally move choice. To balance these objectives, we use the ϵ greedy algorithm. The action is either chosen randomly with probability ϵ or is chosen with $a_t \leftarrow \arg\max_a Q(s_t, a_t)$ with probability of $1 - \epsilon$. To account for the presumably better estimates for the optimal action as time proceeds, ϵ is allowed to vary from a starting value of $\epsilon = .7$ to a final value of $\epsilon = .25$ as the algorithm iterates over the number of games played. This allows for increased random exploration initially, but then goes to a more optimal playstyle as our state data starts to converge.

III. Results

Overall, the chess engine was quite successful in being able to win a King and Rook vs King endgame. It is successfully able to play a randomly seeded Rook and King vs King endgame to a win most of the time. Running the $Q(\lambda)$ learning procedure for 100000 games allowed for the necessary state exploration and learning required for the engine to make action decisions leading up to wins almost all the time. This can be seen in 4 that the cumulative number of wins increases very quickly as we keep acquiring more accurate Q values.

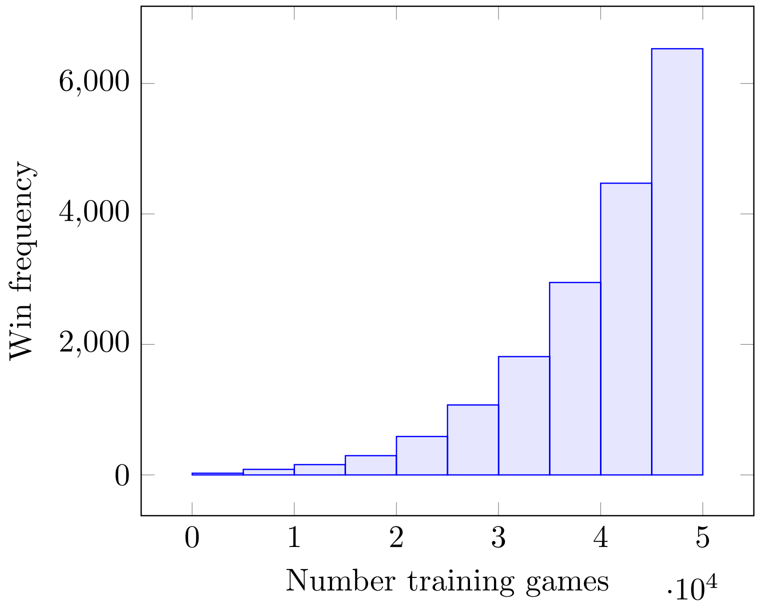

Figure 4. The number of wins acquired during training with 50000 games

To better test the engine we will have it make its optimal decisions in random start positions to reach terminal positions against a random moving opponent. Rather than using the ϵ greedy move choosing strategy above we now have the agent make the optimal action based on the previously trained Q: $a_t \leftarrow \max_a Q(s_t, a)$ at every time step until a termination position is reached. The results of these test matches are shown in table 1. Here we see that just blindly applying Q learning does little to solve this problem mainly due to the massive state/action space that must be covered; even though it hits some reward end states it is not able to propagate these rewards to neighboring previous states well and thus is largely ineffective. Additionally we can see for the 5000 game training case just using the state symmetry method described above increases our win rate by a factor of 7 due to the huge state-space size reduction.

1

Training Method	Training Games	Test Game Wins	Test Game Draws	Win Ratio
Random(baseline)	N/A	7	993	.007
Q no eligibility trace	5000	27	973	.027
$Q(\lambda)$ no state symmetry exploited	5000	103	897	.103
$Q(\lambda)$	5000	740	260	.74
$Q(\lambda)$	20000	879	121	.879
$Q(\lambda)$	50000	985	15	.985
$Q(\lambda)$	100000	998	15	.998

Table 1. Method performance comparisons on 1000 random test games

Another criteria that is desired for this chess agent to complete its task is to examine how many moves it takes to reach a win state. There is a condition in chess called the fifty move rule in which a draw is declared if there are no pieces captured within the last 50 moves. Although this engine is not checking for this condition explicitly, the discount factor γ in the Q learning model ideally should encourage the agent to attempt to end the game early rather than later. We can see that this is partially successful in [5] where the agent wins games with a median of 27 moves with a very small high value tail. However we can see that the agent is not completely successful as there are still quite a lot of games above the 50 move count. Training with a lower discount factor of $\gamma = .7$ and with 100000 game samples we get the distribution to move further to the left with median 13 moves which is a much better move count range seen in: [6]!

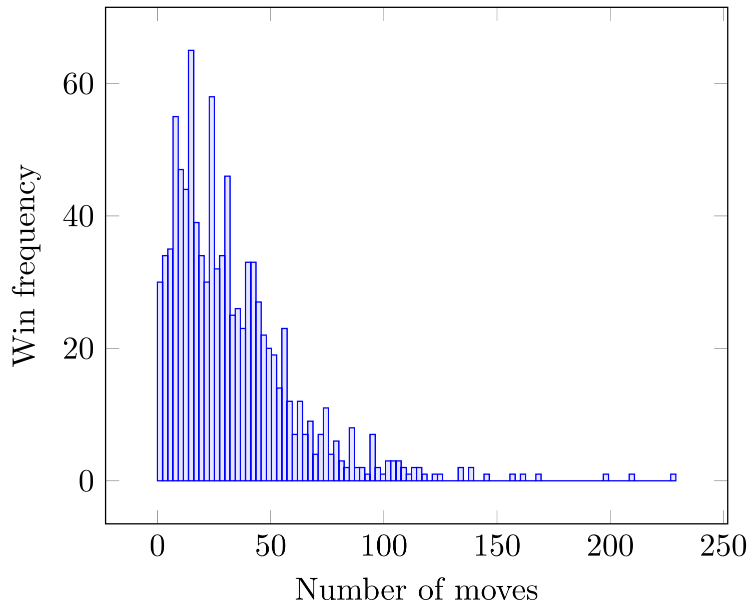

Figure 5. Number of moves to win $\gamma = .9$ 5E4 samples

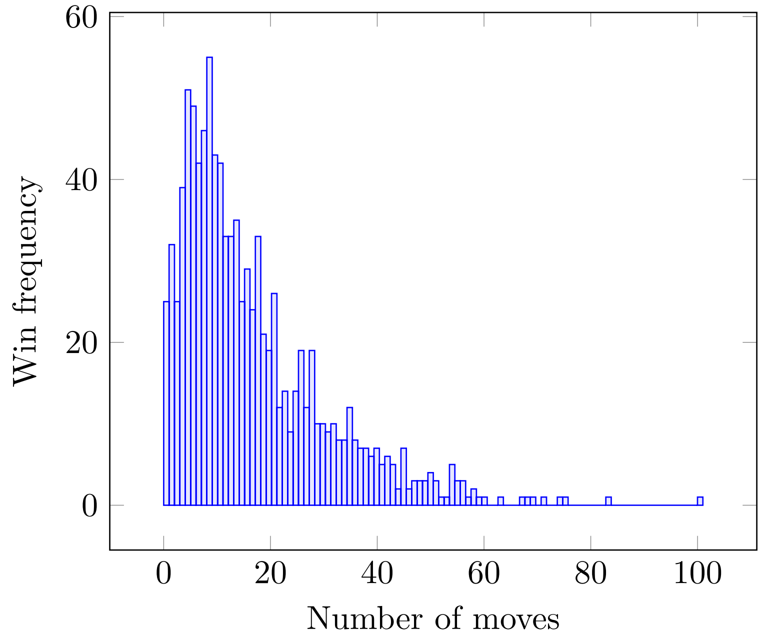

Figure 6. Number of moves to win $\gamma = .7$ 1E5 samples

Lastly a point of interest is to actually intuitively examine the agent's move choices to see if it is actually making reasonable decisions. Looking at one basic randomly seeded position [7]

1. Kc3 Kb1 2. Ra7 Kc1 3. Ra1

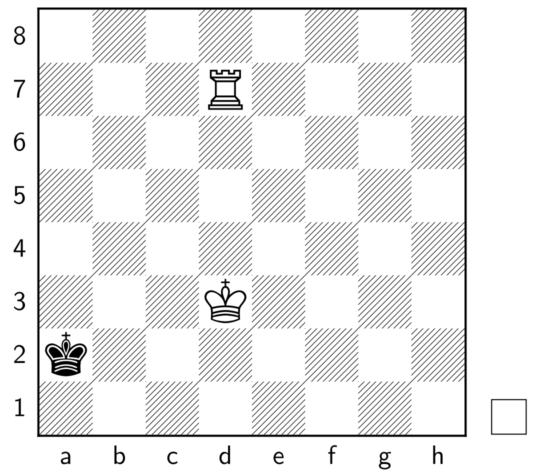

Figure 7. Starting position and moves by engine

we see that the agent is able to find one of the possible forced wins in three moves which is an optimal action for this position! However when going to more complicated positions it becomes apparent for an experienced player to see that the even though the chess engine is able to win most of the time, it is not picking the optimal moves and will often take roundabout methods to a winning a game.

IV. Conclusion

This paper examines the usage of $Q(\lambda)$ reinforcement based learning to teach a computer agent to play and win canonical Rook and King vs King chess positions. After enough training, the engine is able to very consistently win any randomly seeded position of this form. Despite having great performance, the engine still has room to improve as it is not always playing optimally for all positions. One remedy for this could just be to train for a longer interval to explore the state space more and converge Q even more. However, there are likely some smarter techniques for approaching this.

A possible way to try and improve the engine could be to take a closer look at exploiting board similarity. Although we take advantage of axial symmetry to reduce the state-space by a factor of four in this problem, there are many other similar positions that could potentially be grouped together even after this first reduction to form a local/neighbor approximation system for board states. For example, previous studies have looked into categorizing a chess position in terms of "features" as characteristics rather than the actual piece locations.[1] This would allow for fewer Q updates and make it feasible to easily run more games for training giving even better results.

Another interesting task would be to train similar type agents for other more complex canonical chess endgames such as King and two Bishops vs King positions which further increases the state and action space greatly with the addition of another piece. Due to the shear size of state and action spaces for chessgames it seems unlikely an approach like this could ever be used for a full game, but nevertheless such techniques can still be interesting to successfully apply on subproblems within the game.

Appendix: Chess Information

For notational convenience this paper uses standard chess algebraic notation to indicate where a chess piece lies on a chessboard and where it is going to move. The notation is a string of the form:
"Chess piece initial" "column letter" "row number"

For example this location of a piece is described by "Kd5" as the King is in the column "d" and row 5.

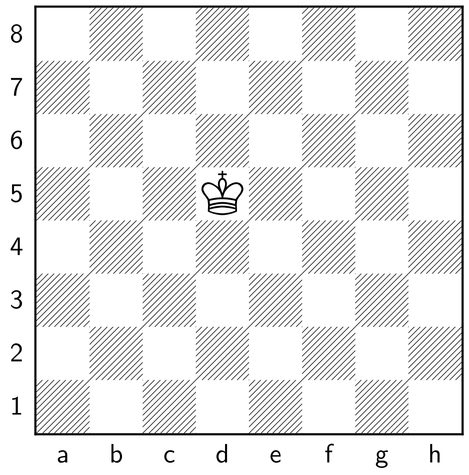

Figure 8. chess position with "Kd5"

The main pieces used in this paper are the King and Rook(also referred to as a castle). The king has one square mobility in any direction while the rook can move as many squares (without jumping) in the horizontal or vertical direction.

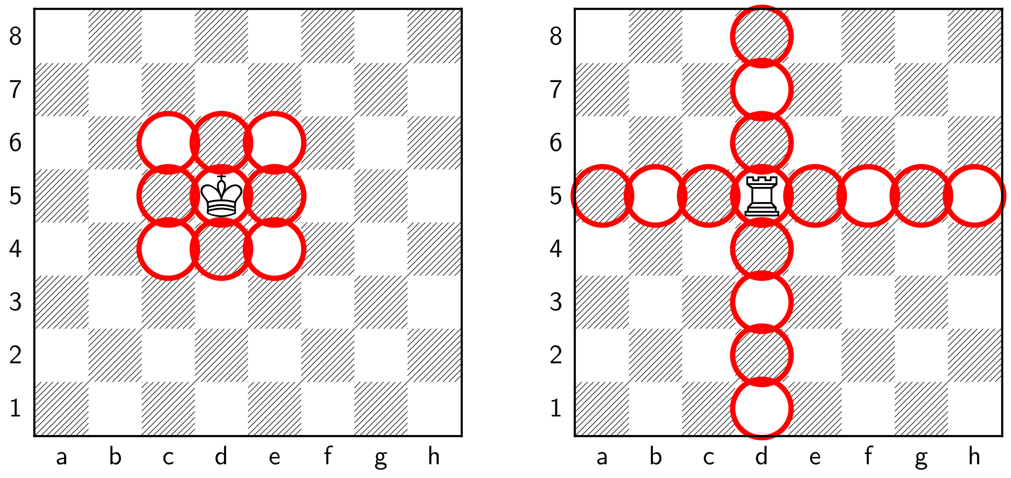

Figure 9. King and Rook movement illustrated

Appendix: Source Code

All source code used for this project is available at: https://github.com/sunild93/RLChess The code is divided between two files: "chess.jl" which includes the implementation of game state and action handling and "Project.ipynb" where the reinforcment learning is carried out.

Acknowledgments

Would like to thank Professor Kochenderfer and the AA228 course assistants for doing a great job teaching the class!

References

[1]Block, Marco , Bader,Maro et al. Using reinforcement learning in chess engines. Research in Computing Science. 2008

[2]Dazeley, Richard Peter. Investigations into Playing Chess Endgames using Reinforcement Learning. University of Tasmania. 2001

[3]Lai, Matthew. Giraffe: Using Deep Reinforcement Learning to Play Chess. Imperial College London. 2015.

[4]Shannon, Claude E. ”Programming a Computer for Playing Chess.” Computer Chess Compendium: 2-13. 1950.