Reinforcement learning project

Conor, Worthington
cw2284@bath.ac.uk

Huan-Hsu, Yeh
hhy36@bath.ac.uk

May 8, 2020

1 Introduction

Chess has always historically been used as a challenge for assessing the capabilities of AI. It was hailed as a milestone when deep blue beat Gary Kasparov in 1996, a time not so long in the past. The reason for the significance of this defeat is that chess is a notoriously hard game for a computer to play, especially as traditional tabular computation methods to defeat the game are almost impossible as a result of their being 10^{40} possible positions and 10^{120} possible games [1].

As such this seemed like an apt challenge for a machine learning project. With current advances in computer hardware and machine learning techniques this problem is nowhere near as intimidating as when first attempted. Furthermore with recent developments in reinforcement learning the current highest performing chess engine is an engine called AlphaZero an engine based around reinforcement learning [2]. Chess like many other games has skill assessed with ELO, whilst our engine will not be competing with a variety of engines, ELO is a very useful measure for assessing how a player or engine performs relativistically to other players.

As such we wanted to experiment with reinforcement learning for the purpose of building a chess AI - roughly symbolic of an average player and thus being able to display some good traits whilst playing but ultimately be able to easily defeat a weak opponent solely by reinforcement learning.

2 Formulating a solution

When deciding on an approach we first agreed on a foundation to base our project. This was a clearcut decision, the leading API for both playing chess and developing engines is from a website chess.com - we utilised their python library as the basis for our engine to provide legal moves and board management.
At this point we then had to decide on the most appropriate way of building a reinforcement model to learn how to play chess. At first we experimented with deep Q learning. It became clear upon research that whilst live engines had used this model to try play chess someone had successfully used it to build an engine[3]. However this implementation required a bit of trickery, as previously

stated the state space for chess is vast and as such required a network trained on supervised data to feed in layers of information pre-processed in an attempt to get the Q-Learning algorithm to work. After reading around this we determined this was not a suitable approach as it seemed overly complex and in this instance produced very poor performance for the work put in.
Another consideration was to try build on the existing research of the leading engine in the world built on reinforcement learning, AlphaZero. AlphaZero is based upon a series of evaluations performed by a vastly complex deep neural architecture which is trained via self play and playing with other leading networks. The network then makes use of a Monte Carlo Tree Search (MCTS) which uses elements of randomness to traverse through potential board options in an optimal fashion. AlphaZero is exceptionally computationally intensive to train and with only a single GPU to train on in a limited time frame this was not an option [4].

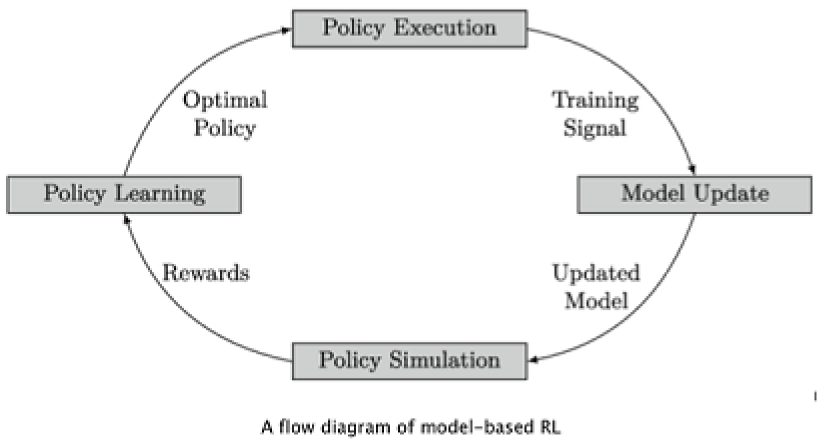

Figure 1: Depiction of model based learning [5]

The final technique we considered to utilise model based reinforcement learning. In this technique we simulate the board and traverse and take samples of board performance to assist in training before execution. This ultimately was the approach we went for as this approach is easily compatible with a lot of conventional techniques used to assist with Chess engine performance and the model for chess is already provided by the python-chess library. As such we opted to utilise this approach and combine it with a deep learning model utilising keras. This technique requires both a method for evaluation of chess boards to evaluate the state of boards so that rewards can be appropriate while training the neural network. Secondly a method of moving through game states whilst simulating chess will be required to enable optimum learning to take place.

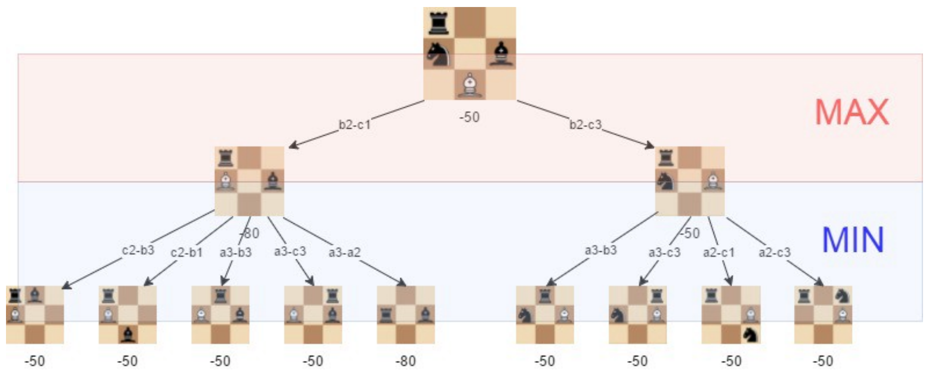

Figure 2: Depiction of minimax search of chess boards [6]

One of the algorithms considered to be used to traverse through all the possible game states is Minimax. With regards to chess we consider the evaluation of the board with regards to all possible moves and then of our best moves we consider our opponents best responses and select the move with the worst outcome for our opponent. Thus maximising our advantage over them. This algorithm is notoriously inefficient as the number of possible moves grows exceptionally quickly as depth increases to the point it becomes unmanageable after a few layers.

Alpha-Beta pruning is a minor tweak to minimax which dramatically improves performance. The algorithm keeps track of two variables alpha and beta, which keep track of the scores of both the player we intend to maximise and the player we intend to minimise. This technique allows for large sections of the tree to avoid being searched without overlooking any better moves. Further if a move seems good it is investigated deeper and if it leads to a large material loss - it can be avoided. This ultimately is the search technique we opted to utilise. With the addition of a Quiescence search to avoid the horizon effect which is when a depth search ends and potentially ignores immediate ramifications a level below where the search ended. A move without a Quiescence search could take material but inadvertently offer check to the opponent the next turn. As such Quiescence search investigates deeper to ensure the terminal nodes of the search are as good as they seem before returning the best state.

With a technique for searching possible games, an evaluation technique to decide how beneficial a board is must be decided upon or else the search can not work nor can the labelling. One potential technique is the oldest technique of evaluation dubbed ‘centipawn’ - this technique attributes values to various pieces of material relative to the value of a pawn such as a rook being worth 10 pawns. This is used to dictate balance as losing certain material such as a queen could cost the game. However this does not take into account vastly more important

factors such as board control and positioning as good players will often concede material to gain dominance over the board and lock down regions.
Another approach for evaluation is to make use of an existing world class engine. The current best engine which is readily available to plug and play with our chess library is stockfish[7]. This engine can easily give a valuation to the game balance with regards to a multitude of factors based on complicated heuristics used to power the game engine. This gives a much more precise reading of score and if learned perfectly by our network would allow for performance equal to a very competent engine. With the ways of training the network planned the design of the network was the next concern. For the network one existing piece of work was utilised which similarly took chess board inputs to produce a prediction of quality as a scalar although this prediction was simply a guesstimate of which side was likely to win - ours is a qualification of how good a board is at any given point for our side. As such we modelled our network in a similar fashion by stacking LTSM layers with dropout. We found with experimentation an additional dense layer also helped performance and the activation function was found to best be suited with a scaled exponential linear unit function. This was probably the result of the value of an imminent checkmate being so much higher than a typical strong board position that this function provides a very accurate map with training[8].

3 Performance

With regards to performance - when building a dataset it's not uncommon to get up to 10k samples from which to learn from a single game due to the amount of paths produced during the alpha-beta pruned minimax. As a result the datasets from each game to learn from are colossal however the loss seems to get reduced very nicely and goes asymptotic after about 64 epochs.

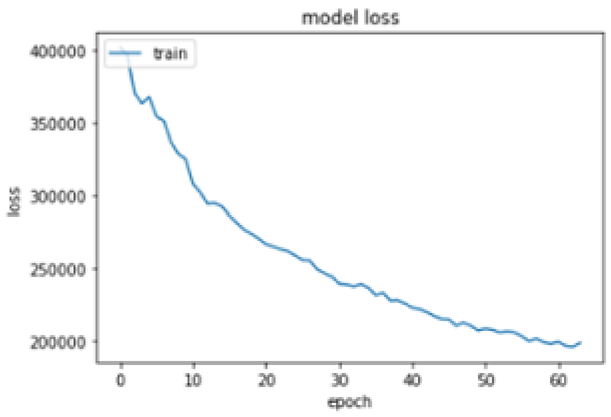

Figure 3: Depiction of model loss over time

This leads to the overall game being played reasonably well. When searching using our Alpha-Beta algorithm twinned with this accurate neural network predictor future move states are reasonably well anticipated as such a number of good chess qualities emerge.

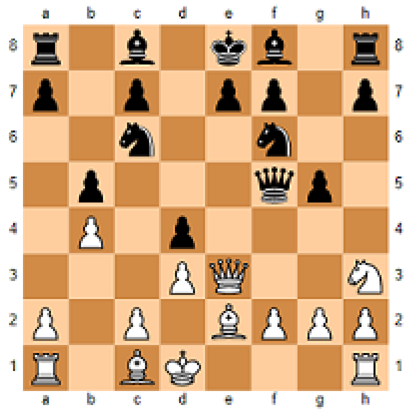

Figure 4: A game being played by a trained agent (Black)

As seen a large amount of material in the image is covered by another piece, there are multiple threats at once and pieces which should be developed last are being developed last. This means no unnecessary moving of the king - a worrying fault which would occur frequently in early learning. All of this was done in 10 games by training off of a set of data we built after each game consisting of every potential move the engine could have made and the score stockfish had associated with that boardstate.

4 Issues encountered

There were a number of issues with this that do still need addressing. For one the engine is incredibly slow - to train and optimise this network is a full working day if not more with an incredibly fast computer. This is mostly as a result of the dataset building and move evaluation being written in serial code and not optimised to utilise the GPU kernel as is done to accelerate the machine learning. Such an optimisation would dramatically increase how much you could train this software and thus test or optimise it. Further specific optimisations or moving to a more efficient algorithm than Alpha-Beta pruning may be beneficial.
Fundamentally this engine does not play the optimal openings and has poor endgame. Openings are very well known in chess and as a result are easy to know what is and isn't good practice - from experience the way the engine plays results in early blunders which certainly weaken it. Furthermore the late game

the engine plays is not particularly good either - I suspect given more testing and more rigorous circumstances it has a tendency to stalemate.
Another issue is as the engine does not know the value of some pieces it occasionally plays certain pieces a bit too actively for too little reward. This includes moving the queen into precarious situations with little to ensure it is not capitulated.

5 Possible improvements

There is a lot of space for improvement currently. As the field of chess engines is a very well researched field one of the first things to consider is how other engines perform so well when searching through possible boards. In one paper a multitude of potential solutions which could provide great benefit to our engine are provided. The first of these is a transposition table which stores positions which have previously occurred to avoid re-evaluation of equivalent positions. The second suggestion is move ordering - in which we would order moves by expected quality and evaluate the best of these to a deeper extent allowing for greater evaluation of more likely moves [9]. Another improvement would be to utilise move databases. With chess there are few good openings and as a result, openings are generally limited in engines to a move database of potential moves the engine can make which will setup resources in the most optimal way to be developed later on. Furthermore at the final stage of the game or 'endgame' there are very few resources and successfully having them work together is a distinctly different style of play so lots of engines alter their play style as the game begins to draw to a close. This can consist of an entirely different engine being used just to solve the endgame. The final improvement would be to potentially move away from the minimax search and instead move towards a monte carlo tree search. As seen from AlphaZero MTCS is the best search algorithm for chess boards and would likely elevate our performance greatly.

6 Conclusion

In conclusion, the model we built can successfully teach itself to play chess. It can easily beat the random opponent and likely performs at the skill level of a decent novice and as such would provide one a decent challenge. The shortcomings could be addressed but ultimately would likely risk either plagiarising AlphaZero or moving the problem away from being a reinforcement learning problem and towards something more heuristic.

References

[1] Rasskin-Gutman, Chess Metaphors: Artificial Intelligence and the Human mind, MIT Press, P205, February 2010

[2] Deepmind, AlphaZero: Shedding new light on chess, shogi, and Go, December 2018, Blog post, Accessed at https://deepmind.com/blog/article/alphazero-shedding-new-light-grand-games-chess-shogi-and-go on 6th May 2020

[3] Groen Arjan, Reinforcement Learning Chess 3: Q-Networks, Kaggle, Tutorial, January 2020, Accessed at https://www.kaggle.com/arjanso/reinforcement-learning-chess-3-q-networks on 6th May 2020

[4] Foster David, AlphaGo Zero Explained in one diagram, Medium, October 2017, Accessed at https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0 on 6th May 2020

[5] integrate.ai, What is Model-Based reinforcement learning, Medium, October 2018, Accessed at https://medium.com/the-official-integrate-ai-blog/understanding-reinforcement-learning-93d4e34e5698 on 6th May 2020

[6] Hartikka Lauri, A step by step guide to building a simple chess AI, freecodecamp, March 2017, Accessed at https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/ on 6th May 2020

[7] iChess,Top 5 Best Chess Engines of the World in 2018, April 2018, Accessed at https://www.ichess.net/blog/best-chess-engines/ on 6th May 2020

[8] Bernhardsson Erik, Deep learning for... Chess, November 2011, Accessed at https://erikbern.com/2014/11/29/deep-learning-for-chess.html on 6th May 2020

[9] Mannem Henk, Learning to play chess using reinforcement learning with database games, October 2003, Masters Thesis, Utrecht University, Section 5, Accessed at https://tinyurl.com/yae92e78 on 6th May 2020