A Reinforcement Learning Approach for Solving KRK Chess Endgames

Zacharias Georgiou [a] Evangelos Karountzos [a] Matthia Sabatelli [a]
Yaroslav Shkarupa [a]

[a] *Rijksuniversiteit Groningen,
Department of Artificial Intelligence
Nijenborgh 4, 9747 AG Groningen
The Netherlands*

Abstract

In this paper we show how a Reinforcement Learning approach can be successfully applied in chess. This is done by focusing on KRK endgames and by implementing the Q-Learning algorithm with different exploration policies. The main goal of this research was to train an artificial agent able to win the endgames as a White Player against an ”experienced” Black Player.

1 Introduction

Since the moment computers and intelligent machines have become able to compute complicated calculations, one way of testing these abilities, has been identified in making them play games. Nothing better than games, in fact, is able to test if a machine is performing in an intelligent way or not. By choosing correct and appropriate actions, its resulting behavior may in fact lead to a win or not. A game that has intrigued scientists during the centuries is chess. This general concept finds concrete evidence already during the 18th century when the Hungarian inventor and engineer Wolfgang Von Kempelen, designed the first chess playing automaton [3] known under the name of ”The Turk”. This machine was designed with the purpose of playing chess in a completely autonomous way, without any human help it was in fact, supposed to beat almost all of its human opponents. This early and rough robot turned out to become a celebrity, not only in Europe but also in the United States, its inventor in fact toured between the old and the new continent, making his creation play against very famous opponents of that time. One of the most important games that may be found in chess literature was played by ”The Turk” against Napoleon Bonaparte, where the french general lost very quickly in not even 25 moves. However, this machine turned out to be a deceit, Von Kempelen’s invention wasn’t in fact able to play autonomously at all, inside the machine there was place for a human person that was able to see the chessboard and move the robot with some commands, without being seen from the audience. Although ”The Turk” can’t be defined as an authentic autonomous player, since it turned out to be a fake, it still has the merit to be the first example in which machine and humans challenge each other by playing chess.
The idea of building a machine able to beat the best human players at chess has inspired several mathematicians and computer scientists during the years and the moment they were all waiting for, arrived in May of 1997 [5]. Deep Blue, a chess playing software developed by the American company IBM, was able to defeat the current chess world champion Garry Kasparov in a six-game match with the score of 3.5 against 2.5. It was the first time that a chess A.I. was able to defeat the best chess player in the world, this event is considered as an absolute breakthrough and represents the end of an era for chess players and the start of a new one for A.I. scientists.
However, although A.I. has reached such an important result, chess still remains one of the most intriguing dilemmas that researchers are trying to solve. If at the one side, building a program able to beat the best human players of the world isn’t a challenge any more, solving the overall game by estimating all

the possible combinations of pieces on the board and asserting if a certain position is winning or not, still remains a big question mark. This due to the fact that chess is part of the NP class of problems [6] and computing all the possible combinations needed to solve the game is out of nowadays computational resources.
The main purpose of this paper isn't of course, solving this thousand-year-old game, however we try to add some knowledge to this constantly evolving research field by investigating a machine learning technique, that deals with reinforcement learning. This kind of approach is used to train a chess agent in order to make him able to play a particular type of chess end game called the KRK one. In this type of position, only 3 pieces are left on the board: the White King, the White Rook and the Black King and the main purpose for the white player is to checkmate its opponent by coordinating the movement of its 2 pieces. In the next sessions, after a brief state of the art part, our approach for training the agent will be proposed together with a more detailed explanation of the scenario presented on the chessboard.

2 Related Work

Although Deep Blue's win against Kasparov is considered as a breakthrough in Artificial Intelligence, the way IBM's super computer played the match is far away from how human chess masters play the game of chess. In fact, Deep Blue had almost no understanding of what was going on on the board, the only way it was able to choose the correct moves at each turn, was by calculating multiple possible combinations very deeply. This kind of approach based on a "brute force" strategy is known as Weak A.I. Researchers have so tried to develop more human likely programs that are able to have a more deep understanding of the game, without only basing their decisions on pure calculations. In order to pursue this goal, one strategy has been identified in Reinforcement Learning, more in detail in what is called Temporal Difference Learning. One research that deals with it may be found in work [2], here the authors present their own chess program called KnightCap that combines a variation of TD Learning called TDLeaf with a game tree search approach. The main objective of the program was to learn an evaluation function that governed its playing strategy. This function has been learned by making the program play online against human players. The results were pretty good, in fact in only 300 games KnightCap was able to increase its ELO rating up to 2150 points, which would make him almost as good as a National Master titled player. However, this rating if far away from the top player's one, in fact a Grand Master has over 2500 ELO points. The reason why KnightCap couldn't increase its rating more may be identified in its evaluation function. As explained in the paper the only parameters that the engine uses for estimating a position on the board regard the amount of pieces that each player has. As a result, the moves it produced didn't involve tactical errors, which means giving pieces away for free, but were completely ignorant from a strategical point of view. Although an implementation of the TDLeaf algorithm produces some good results the understanding of chess is far away from the chess engine presented in the paper.
A work that takes into consideration the lack of parameters that characterizes the evaluation function of the previous one is [4]. In order to face this issue a Neural Network has been used to produce the function, once this is built the chess engine is trained by using TD Learning on a database of games played by high rated players. Here much more parameters are considered, that make the understanding of the game of the chess engine closer to the one of more expert players. For example, particular attention is not only put on the material that is present on the board, but also on how it's used. This aspect is related to concepts like connectivity, which means how many pieces a piece is able to defend, an aspect very important in endgames. Moreover also the activeness of the pieces is considered, that is, how many squares they are controlling. Taking this parameter into consideration an active Knight may be much stronger then a trapped Rook. The results of this work show that the chess engine was able to evolve appropriate evaluation functions to make him a strong player, in a short period of training.
These are two examples of researches that deal with the game of chess in general, which means trying to make an artificial agent able to play the whole game. Although we would have liked to research this kind of approach due to time issues and computational resources we had to focus on a smaller research question by considering only one particular aspect of the game. However, although our goals and the ones presented in the papers may be different, they still have been inspiring and have guided us in this paper.

3 Our Approach

As already mentioned in the introduction, the main goal of this work was to train a chess program in order to make him able to win a KRK endgame. Before starting to investigate into detail the approach that has been used in order to pursue this objective, it's important to understand the board situation that the agent had to deal with. A KRK endgame sees only 3 pieces left on the board, a Rook and a King for the White player and only the King for the Black one. This endgame is defined as theoretically winning, which means that the White agent is able to win all the possible situations presented on the board. This happens because, if it's able to coordinate the movement of its 2 pieces correctly, it will always be possible for him to cover enough squares of the board in order to checkmate its opponent, no matter how good this last one will play. It's important to assert that this isn't true for all type of endgames, cases in which the White player has only one Bishop or a Knight instead of the Rook turn out to end in a draw. An example of a KRK endgame is presented in the figure hereafter:

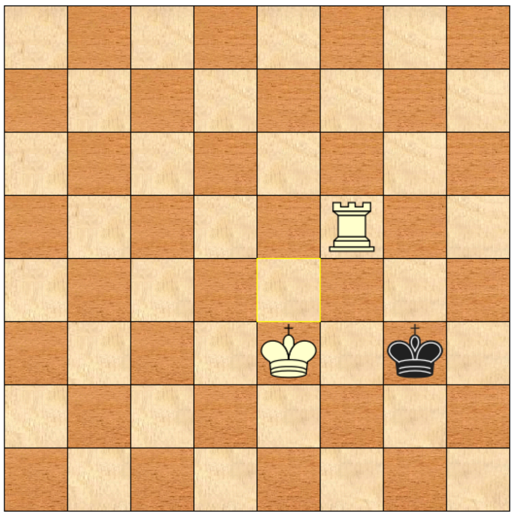
Figure 1: An Example of KRK Endgame

The training method we have used is part of a particular subset of Machine Learning techniques that are known under the name of Reinforcement Learning. As explained in work [1] Reinforcement Learning algorithms are used when the main goal of training an agent, is making him produce a correct set of actions. This set of actions has to lead to a particular final state in which the agent will receive a reward for having been able to reach it. Single actions aren't however important enough to gain the reward, what's crucial, in fact, is that the agent should be able to identify a correct sequence of actions that lead him to the desired final state. This chain of actions is called policy. Due to the importance of policy, Reinforcement Learning is very often used in game playing, in fact, nothing more then games are a concrete example that are able to highlight how less important a single move may be, but how crucial it can turn out if it's part of a correct sequence. The method we have used in order to train the White player was by implementing the Q-Learning algorithm, which is elaborated more in the next subsection.

3.1 Q-Learning

In Q-learning the model consists of multiple states, let them be S, and every state contains multiple actions that the agent(s) can take, let the possible actions be A. The agent can only take an action such as $\alpha \in A$ in order to move from the one state to the other and each action provides the agent with a reward based on how effective the move was towards the completion of the ultimate goal. The reward, represented as r, consists of the weighted sum of the reward of all future steps starting from the current state where the weight for a step from a state Δt steps into the future is calculated as $\gamma^{\Delta t}$. γ is the discount factor, which determines the affect that future actions have. For example high γ value gives more importance to actions closer in the future while low γ values tend to distribute the value more evenly to both close and distant actions (although actions in the future still have lower weight). Both the learning value α as well as the discount factor γ need to be within $0 < \alpha, \gamma < 1$. Initially the value for every state (notated as Q) is arbitrary set by the designer. Having defined all the above, we can move on and say that in each training iteration (epoch) the formula 1 is executed. In equation 2 we see a more explanatory version of equation 1.

$$Q(s,a) \leftarrow Q(s,a) + \alpha * [r_{t+1} + \gamma * max_{\alpha} Q(s_{t+1}, a_{t+1}) - Q(s,a)] \tag{1}$$

$$\overbrace{Q(s,a)}^{NewValue} \leftarrow \overbrace{Q(s,a)}^{OldValue} + \overbrace{\alpha}^{Learningrate} *[\underbrace{\overbrace{r}^{Reward} + \overbrace{\gamma}^{Discountfactor} * \overbrace{max_{\alpha} Q(s', a')}^{Optimalfuturevalue}}_{Learnedvalue} - \overbrace{Q(s,a)}^{Oldvalue}] \tag{2}$$

It is important to highlight that we have trained two agents, both the black and the white players, since they have different goals and relative end states. For the white an end state is when he wins (he gets there by making a move that checkmates its opponent), this is rewarded with the value of 1, while if the game results into a draw it gets a reward of 0. On the other hand, black's end state is when the game results into a draw. In that case the black agent gets a reward of 1 while if it gets checkmated its reward is 0. According to the rules of chess, a game ends in a draw if the white agent hasn't been able to checkmate its opponent within 40 moves.

3.2 White/Black states problem

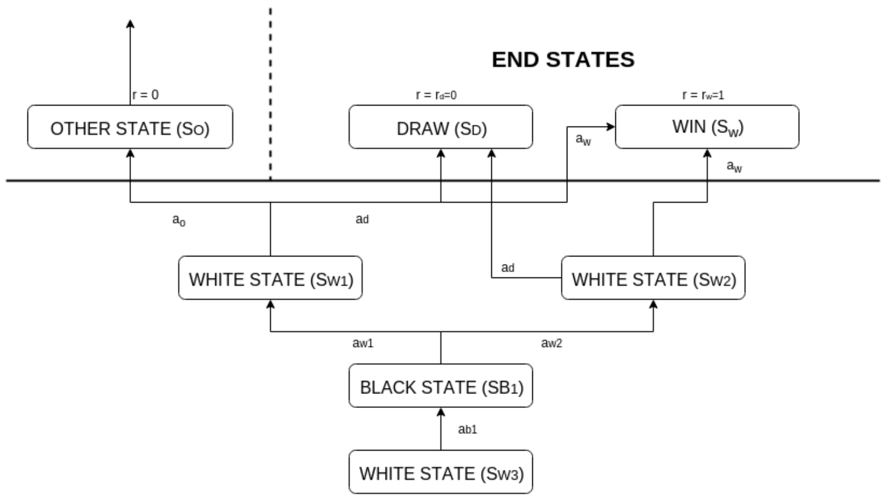

Figure 2: States diagram

The major problem we faced with our approach was the changing of turns between the black player and the white one. Our main goal regarded training the white player in order to make him win. Considering Q-Learning this basically means simply reaching winning state. Unfortunately, due to the chess rules, there is a switch between the black and the white moves (states) when both want to reach the goal. It is not possible to do it in straightforward way simply by calculating Q scores for states in the same way for black and white players (agents). If it is done in such a way it turns out that the black player is simply helping the white one to win while the agents are training. Such an unpleasant behavior happens because during the training the Q value is calculated assuming that the black player will pick up the worst move for him (which is actually the best one for white player). However, as it happens also in real chess, we should assume that the opponent makes the best possible move for himself and of course not for the other player.
In order to solve this problem we decided to implement a slightly different strategy to update the Q scores of the players. In order to illustrate the strategy lets refer to diagram to Figure 2. Let's assume

that the agent is at a black state S_{b1} and selects action a_{w2} that moves him to the white state S_{w2}. To calculate the Q value for this action, we have to assume, that in next move the white player will do the best possible move available (the move has the highest Q score) which in this case corresponds to a win (checkmate state). The general update rule for the black moves can be found on 3. In this example to calculate the Q score for S_{b1} state and a_{w2} action the equation becomes 4.

$$Q(s,a) \leftarrow Q(s,a) + \alpha * [r_{t+1} + \gamma * max_{\alpha} Q(s_{t+1}, a_{t+1}) - Q(s,a)] \tag{3}$$

$$Q(S_{b1}, a_{w2}) \leftarrow Q(S_{b1}, a_{w2}) + \alpha * [0 + \gamma * Q(s_{w2}, a_w) - Q(S_{b1}, a_{w2})] \tag{4}$$

Lets now consider the white state presented in example S_{w3}. Suppose it has chosen action a_{b1} which leads to the black state S_{b1}. In case of a white state we need to assume that the opponent will pick up the best move available for him. The best move for the black player is the worst for the white player, in or case the move with the lowest Q score. So to update Q score for white moves we used update rule 5.

$$Q(s,a) \leftarrow Q(s,a) + \alpha * [r_{t+1} + \gamma * min_{\alpha} Q(s_{t+1}, a_{t+1}) - Q(s,a)] \tag{5}$$

3.3 Exploration policies

Initially we have run our experiments with a random exploration policy, that is, the agent moves randomly from one state to another until he reaches an ending (winning or draw) state, let it be S_{end}. When that happens a reward is given to the action that got him to that state. In the next game the agent will still move randomly, even if the option to preform an action that previously gave him a reward is available. Thus from state S_{end-1} there is no guarantee that the agent will go to state S_{end} but if he does, the Q value of that state Q_{end} propagates and the action $s_{end-1} \rightarrow s_{end}$ gets proportion (the exact value depends on the γ and α rates) of the reward. After many such iterations, and statistically speaking, the agent is bound to explore all possible states ($64 * 63 * 62$ in our case since we are using three pieces).

This policy however turns out to be very time consuming and a lot of training is required, therefore we also tried to ϵ-greedy policy which, ideally, would lead the same results with less computational time. The reasoning behind this relates to the fact that the agent, instead of making always random actions, only has a certain probability of doing so, this probability is defined as ϵ. For example, if the agent currently finds itself in state S_n and the next possible actions are $A_{n+1,n+2,n+3}$ the agent has ϵ chance of picking one of these actions randomly, and $1 - \epsilon$ of picking the action with with the highest Q value.

4 Results

4.0.1 Winning moves

One way to measure whether or not our approach works was to count the winning game's rate. This however, although fairly accurate, was not sufficient, since just by allowing agents play forever, it could result a win. In order to eliminate this possibility we put an upper threshold of moves (40 in total) after which the game would result to a draw. In top of that we decided to also measure the average number of moves a game would last, and not surprisingly our agents managed to win with an average of 10 moves.

4.1 Discount and Learning rate and epsilon parameters

In this section we present the results that we gathered from our experiments. As mentioned above we trained our agents with Q-Learning. During the learning phase of the agents, the learning rate α was kept constant while we experimented with various discount values γ. After various γ configurations we concluded that, an ideal discount value is $\gamma = 0.99$.

4.2 Winning percentage

In figure **??** we compare 2 different ϵ-greedy strategies to the completely random approach. As may be seen from the picture with ϵ = 0.9 there is a faster convergence to the optimal result comparing to the random strategy. On the other side, with $\epsilon = 0.1$ the convergence to the optimal result is slower but the percentage of wins is higher.

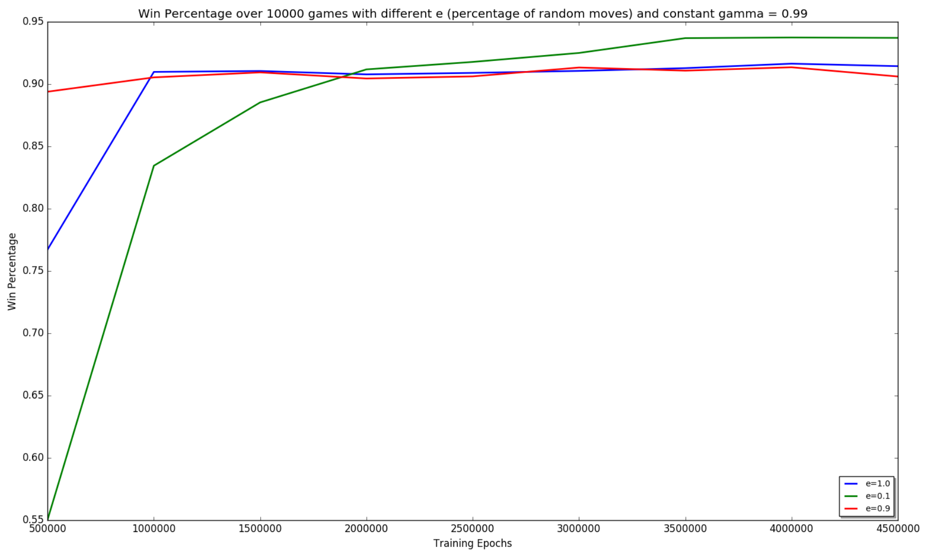

Figure 3: Winning percentage with ϵ-greedy policy comparing to e = 1

In figure **??** we present the average number of rounds that a game needs to be finished. As may be seen the completely random approach together with the $\epsilon = 0.9$ one the number of moves is approximately the same without depending on the number of episodes. On the other side $\epsilon = 0.1$ has a higher average of this criterion, but as already seen in the previous graph it performs better when the number of training epochs increases.

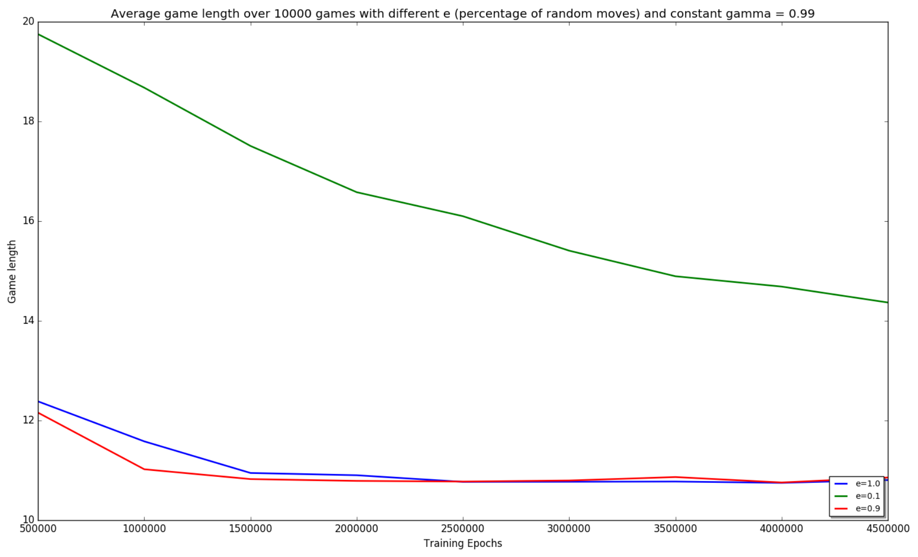

Figure 4: Average game length with ϵ-greedy policy and e = 1

4.3 Training time

We decided to train the agents multiple times with different γ values and different number of episodes and see at what point the training converges to a satisfactory winning percentage. As we can see in figure 5, discount values of 0.4 or more managed to achieve more that 90% winning rate for training less than 5 million episodes while discount values of less than 0.4 did not achieve nearly as good results with the highest of them (0.3) reaching only up to 75% winning rate. While using the ϵ-greedy policy we did not repeat the procedure, instead we just applied the optimal γ resulted from the random policy.

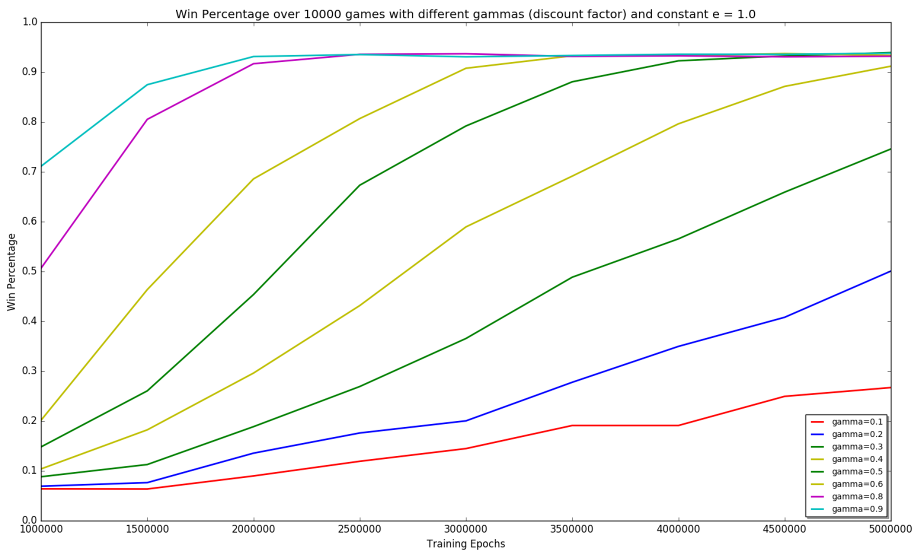

Figure 5: Winning rates based on different γ values.

5 Discussion and Conclusion

Observing the agent play it's possible to see that it has leaned some of the theoretical strategies that are needed to win a KRK endgame. The first one regards using the Rook in order to force the BK always closer to the boarder of the board. Once this is done it's possible to observe that the White Player is able to gain what is called opposition, with its King it's able in fact to avoid that its opponent's one can advance, which would mean giving it the possibility to escape. Moreover, once this opposition is gained the White Player is also able to loose a tempo in order to force the black one, in making whatever move that will lead to a checkmate in the next round. Also the Black Player has identified some possible counter play strategies, it's indeed clearly while observing him playing that it tries to attack its opponent's Rook whenever it has the chance. This is done in order to try to gain some tempos on the opponent and maybe get closer to the 40 moves rule which would lead him to a draw. Unfortunately for him, the White Player has identified the correct strategy to face these threats by simply defending its attacking piece or removing it from its opponent attack. Although we are more than satisfied about the outcome of the project since the agents play in the best way possible and the winning percentage for Q-Learning is above 90% with any policy, there are a couple of things that we would like to improve.
One possible approach that has been tried in order to improve the training time of the algorithm by making it faster that we tried regards the annealing of different ϵ. However, this hasn't produced any successful results, in fact, the winning percentage was approximately the same compared to our main strategy. Moreover we have also tried to implement the Sarsa Learning Algorithm, but the results obtained are far from successful since they only reach up to a 30% winning rate.
However, in this work we have shown how Q-Learning leads to more then satisfying results in training an artificial agent in order to make him win a particular kind of endgame. As human chess players that

start studying the game properly by learning the endgames firstly, we did the same with this very rough and primitive version of potential chess engine.

References

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[2] Jonathan Baxter, Andrew Tridgell, and Lex Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.

[3] Gerald M Levitt. *The Turk, Chess Automation*. McFarland & Company, Incorporated Publishers, 2000.

[4] Henk Mannen and Marco Wiering. Learning to play chess using td (λ)-learning with database games.

[5] Stuart Russell and Peter Norvig. Ai a modern approach. *Learning*, 2(3):4, 2005.

[6] Bart Selman, Hector J Levesque, David G Mitchell, et al. A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446, 1992.