University of Amsterdam

Artificial Intelligence

Bachelor Thesis, 18 EC

DeepChess: Deep Reinforcement Learning applied to the endgame of chess

In this thesis the creation and performance of DeepChess is discussed, the first chess engine that is based on the findings of DeepMind's AlphaGo. DeepChess's performance reached a 88 % winning-rate in winning positions in the King Pawn King endgame against chess engine SunFish. With more experiments and an extension to the full game of chess, I will provisionally conclude that it is possible to apply Deep Reinforcement Learning to the game of chess, and that such a chess engine could perhaps improve on the current chess engines based on brute-force methods.

Author:
Sierk Kanis, 10688528

Supervisor:
Efstratios Gavves

June 24, 2016

University of Amsterdam

Contents

Acknowledgements

I would like to thank my supervisor Efstratios Gavves and my co-supervisors Matthias Reisser, Changyong Oh and Berkay Kicanaoglu for support during this project. Without their support it would not have been possible to achieve the aims for this project within the available time.

1 Introduction

During the last two decades machine readable information has increased exponentially, and with that, so has the need to understand and use this information. A technique gaining more and more interest is Machine Learning, which provides tools with which large quantities of data can be automatically analysed (Hall, 1999), and thereby easing the burden of hand-programming growing volumes of increasingly complex information (Michalski et al., 2013). Machine Learning proved to be helpful in tasks which are difficult to program by hand, such as data mining, machine translation, speech recognition, face recognition and robot motion (Pierre Lison, 1996).

The theory of reinforcement learning provides an account of how agents may optimise their control of an environment. While reinforcement learning agents have achieved some successes in a variety of domains, it could only be used in fields where useful features could be handcrafted (Mnih et al., 2015). In more complex tasks, features can not be handcrafted sufficiently, as is the case of tasks that are so deeply familiar to humans that it is hard to explain their behaviour explicitly, driving a car for example (H. Tsoukas, 2005), or of tasks that are not yet fully examined, example given, just developed drones that should perform flips. To use the concept of reinforcement learning successfully, without explicitly stating features and in situations approaching real-world complexity, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalise past experience to new situations (Mnih et al., 2015).

With the rise of Deep Reinforcement Learning, agents can now successfully learn policies from high-dimensional input data. This is tested by DeepMind in the domain of classic Atari games, where the agent, when only receiving pixel data, surpasses the level of all previous algorithms and achieved a level comparable to professional human gamers (Mnih et al., 2015). Also, the game of Go, long viewed as one of the most challenging games for artificial intelligence, has been tackled with the use of deep reinforcement learning, achieving a winning-rate of 99.8% against all other Go programs and defeating the human World Go Champion Lee Sedol (Silver et al., 2016).

For the game of chess, however, the current best engines are based on brute force calculation of a limited amount of moves, in combination with a handmade evaluation function tweaked both manually and automatically over several years (Lai, 2015). Being inspired by the trend of applying deep reinforcement learning to games, which DeepMind started by achieving superhuman performance on Atari games, a natural question arises: could an agent, which would be able to form more complex evaluation functions, improve on the current chess engines, while being independent and not restricted by any human knowledge? As an attempt of answering this question DeepChess has been created, an engine based on Deep Reinforcement Learning, which has learned to play the endgame

of chess, specifically, the endgame of King and Pawn against King (KPK).

In the literature overview, Reinforcement Learning will be explained, together with its disadvantages. Subsequently, the benefits of connecting a deep neural network as function approximator will be stated, to finally discuss some research within deep reinforcement learning. In the method section, the literature will be applied directly to chess, consisting of state representation, the network architecture and evaluation methods. In the result section, the performance of DeepChess will be evaluated, resulting in winning 88% of theoretically won positions against chess engine SunFish, and winning 99% of winning positions against a random playing opponent. This corresponds, according to Silman's classification of chess endgames, to DeepChess being able to play this endgame with an approximated Elo-rating of 1400 (Silman, 2007). On the basis of these results I will conclude that, with tweaks, DeepChess could be able to play the KPK endgame optimally. As recommendations for future research, the architecture and hyper-parameters of DeepChess could be further optimised, together with an extension of learning a broader variety of chess endgames.

2 Literature overview

In the following section, relevant literature concerning different machine learning methods is discussed, concluding that Deep Reinforcement Learning could be used best as an attempt to improve on the current chess engines. Firstly, Reinforcement Learning is discussed, subsequently, Deep Learning and Deep Reinforcement Learning are described. Finally, previous work on applying Deep Reinforcement Learning to chess is discussed.

2.1 Reinforcement Learning

In this section the basic idea of Reinforcement Learning is discussed, subsequently an application in chess is described and finally the motivation to add Deep Learning is stated.

Tasks are considered in which an agent interacts with an environment; selecting actions, observing states and receiving rewards. The goal of an agent is to interact with the environment by selecting actions that maximise future rewards. A position that covers all information necessary to formulate the best action is called a Markov State. Whenever these states terminate in a final number of steps, a large but finite Markov Decision Proces (MDP) is formed. As a result, Reinforcement Learning methods may be applied to solve these MDP's, resulting in an optimal strategy for receiving future rewards (Mnih et al., 2015).

The goal of Reinforcement Learning is to learn the optimal policy, in other words, the optimal way of acting in an environment. This policy is found by

finding the optimal action-value function Q*(s,a), which pairs each state with the action that maximises the expected future reward. This optimal action-value function obeys the Bellman optimality equation (1), which is based on the following: when the Q values of all actions from the next state are known, Q*(s',a'), the optimal strategy is to select the action that maximises the expected value of r + Q*(s',a').

$$Q^*(s,a) = [r + \gamma max\, Q^*(s',a')|s,a] \tag{1}$$

in which γ is the discount factor determining the agent's horizon, and r the reward the agent receives for performing move a. In the case of solely terminal rewards, the agent only receives a reward when a terminal state is reached. Only at this moment, the agent retrieves the real value of a state. The next iteration, using the Bellman equation, it gives this information back to the one but last state; the next iteration it gives this information to the second to last state, and so one. Eventually, the whole action-value function is known, using this iterative update based on the Bellman optimality equation (Silver, 2015).

The idea behind many Reinforcement Learning algorithms is to approximate the action-value function, using the above stated iterative update. It is proven that such iterative algorithms converge to the optimal action-value function (Sutton & Barto, 1998). However, this basic approach is totally impractical, since all states are visited before reaching this optimal action-value function, without using any generalisation. To this end, it is common to use a function approximator, which is typically a linear function, but a non-linear function such as a neural network could be used instead. The parameters of this approximator are updated in such a way that it optimises the following loss function (2):

$$L_i(\theta_i) = (r + \gamma max\, Q(s',a',\theta_i^-) - Q(s,a,\theta_i))^2 \tag{2}$$

in which θ_i are the parameters of the trainable Q-network at iteration i and θ_i^- the parameters of the target network at iteration i. In contrast to supervised learning, the targets are not fixed before learning, but formed by predictions of a fixed target Q-network. More precisely, every C updates the trainable network gets cloned to obtain the target network, which is then used for generating the targets for the following C updates to the trainable network (Mnih et al., 2015). Using this loss function the network is not tested on its ability to predict the outcome of the game from the current state, but on its ability to predict its own evaluation in the next state. As long as some states receive fixed rewards, the network is claimed to eventually achieve temporal consistency, meaning that sequential states have similar Q values (Lai, 2015).

It may not be necessary to iterate until the minimum of the action-value function is reached, since the optimal policy could already be reached. Therefore, it is more efficient to use stochastic gradient descent. Also, the agent learns

on-policy, meaning that its learning is based on already learned behaviour, which is often selected by an epsilon-greedy strategy: during training the amount of exploration (doing random moves) will be decreased to ensure more exploitation (choosing the best move following the current policy) (Mnih et al., 2015).

In 2001 chess program KNIGHTCAP achieved an Elo-rating of 2500 based on temporal difference learning, which is a form of Reinforcement Learning (Baxter, Tridgell & Weaver, 2001). The value of being in a specific position (the value function) was formulated as a combination of handcrafted features multiplied by their weights, which were learned by an iterative reinforcement process (Block et al., 2008). The algorithm would then repeatedly select the action that results in the highest valued position and would play according to this rule.

However, using handcrafted features has two drawbacks. Firstly, it is dependent on expert knowledge of chess, which is necessary for creating handcrafted features. Secondly, knowledge chess players use is so practical and deeply familiar to them, that it is hard to express this knowledge in words (Tsoukas, 2005). This difficulty of explicitly stating features that characterise a chess position, adds to the disadvantage of handcrafted features. Fortunately, a technique called Deep Learning has risen up recently, which makes the system itself perform feature extraction. The idea of Deep Learning is illustrated in the following paragraphs.

2.2 Deep Learning

To go beyond weight tuning with hand-designed features and actually have a learned system perform feature extraction, a highly non-linear function approximator is necessary. In this section, the idea of Deep Learning is expressed, followed by recent achievements of the technique.

To perform classification directly on pixels, or large input data, a simple model is not sufficient. The system would have to memorise all different input data, which would require high computational power and would not be able to generalise well (Lai, 2015). Bengio (2009) states in the book *Learning deep architectures for AI*: "when a function can be compactly represented by a deep architecture, it might need a very large architecture to be represented by an insufficiently deep one." This claims that a more efficient approach for classifying complex structures is to use a deep architecture, where each layer would identify a different level of features. The first layer would identify *low-level* features like corners and edges, whereas the second layer could use the output of the first layer to identity slightly *higher-level* features such as different shapes. In this way, one could imagine that higher-level abstractions which characterise the input could emerge (Bengio, 2009).

In 2012, Krizhevsky et al. trained a deep neural network for the ImageNet

competition and achieved an error of 15,3%, when the previous best was 26,2%. Deep Learning has also been used for handwriting recognition, where deep networks are approaching human performance with 0.3% error (Krizhevsky et al., 2012). Also, between 2010 and 2014, the two major conferences on signal processing and speech recognition, IEEE-ICASSP and Interspeech, have seen a large increase in the numbers of accepted papers in their respective annual conference papers, on the topic of Deep Learning for speech recognition (Deng & Yu, 2014). These successes have led to a wide usage of deep neural networks in the fields of speech recognition, image recognition, natural language processing and recommendation systems (Bengio, 2009). Recently, the combination of Deep Learning with Reinforcement Learning has gained interest, which will be discussed in the next section.

2.3 Deep Reinforcement Learning

In this section the basic idea of Deep Reinforcement Learning is illustrated, followed by some achievements of this recent combination. Finally, two changes to the original Reinforcement Learning algorithm are stated.

The idea of Deep Reinforcement Learning is to connect a deep neural network to approximate the action-value function used in Reinforcement Learning, thereby, combining Deep Learning with Reinforcement Learning (Mnih et al., 2013). This way the agent is able to learn directly from board positions, without having to implicitly state features as a guide to the learning process. There are two major benefits of this learning method. Firstly, no expert knowledge is required at all, since the agent can come up with features itself. As an example, it is not required to introduce strategic chess concepts, like the distance of both kings to the pawn, the concept of opposition, the distance of the pawn to the queening square, etc., which should have been necessary for basic Reinforcement Learning. Secondly, Krizhevsky, Sutskever and Hinton (2012) state that by feeding sufficient data in the deep network, it is often possible to learn better representations than was possible with handcrafted features.

Tesauro's TD-Gammon (1992) was the first showing the power of the combination of Deep Learning and Reinforcement Learning, by greatly surpassing all previous computer programs in the ability to play backgammon (de Dios, Cajías & Martínez, 1992). Inspired by this success, a novel Q-learning algorithm was published by Google's Deepmind in 2015, that achieved superhuman performance in several classic Atari games, using the same algorithm, network architecture and hyper-parameters. The agent's input only contained pixel data and score information from the game emulator and thereby, bridging the divide between high-dimensional input data and output actions (Mnih et al., 2015). In 2016 Deep Reinforcement Learning was combined with supervised learning from human expert games and Monte Carlo simulation, to pioneer the victory of a computer program over a human professional on the full-sized game of Go (Silver et al., 2016).

However, in *Human-level control through Deep Reinforcement Learning* Mnih et al. (2015) state that Reinforcement Learning is known to be unstable when a non-linear function such as a neural network is used to represent the action-value function. This instability has several causes: the correlations present in the sequences of states; the fact that small changes of the network can drastically change the policy, leading to a change of the state distribution; and the correlations between the trainable network and the target network.

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Figure 1: The final Q-learning algorithm in pseudo-code

To address these instabilities two changes were made to the original Q-learning algorithm. Firstly, a replay memory was added, which stores the last N experiences. Random tuples (s, a, r, s') were picked from this replay memory and were used for training. This breaks the correlation between states and smooths the distribution of the data, avoiding stagnation in local minima. Meanwhile, the agent was playing games to refresh the replay memory. Secondly, the target network was only updated periodically, which adds a delay between the time of an update to Q and the time the update affects the targets, making divergence or oscillations much more unlikely (Mnih et al., 2015), see Figure 1.

2.4 Deep Reinforcement Learning in chess

In this section, the current state of chess computers is illustrated, followed by the problem of evaluation functions in chess. Thereafter, the achievements of chess engine Giraffe are discussed, ending with the difference between Giraffe's and DeepChess's approach.

In 1997 World Chess Champion Garry Kasparov got defeated by IBM's Deep Blue. For the first time in the history of chess, computers proved to be stronger than humans. The strongest chess computers rely heavily on brute-force methods: calculating all possible combinations of moves to a certain depth (Hsu, 2002). How can a human searching 3 to 5 positions per second be as strong as a computer searching 200 million positions per second? Apparently, humans calculate much more effectively and rely on their intuition gained by experience. It is hard to define concrete rules to increase computer's search effectiveness without overlooking strong moves (Lai, 2015).

A problem with most current chess engines is their evaluation functions, which assign scores to positions without calculating further. These functions contain most of the domain-specific knowledge in chess engines. The best chess engine at the moment, Stockfish, has an evaluation function which is designed with the help of many grandmasters, and consist of more than 100 handcrafted features, slightly manipulated over the last few years (Lai, 2015).

As an attempt to solve the problem of evaluation functions in chess, Giraffe was engineered, a chess engine based on Deep Reinforcement Learning. Giraffe is the first successful attempt at using machine learning to create a chess evaluation function, with minimal hand-coded knowledge. It reached a level of International Master (Elo-rating of 2400) and achieved at least comparable positional understanding compared to the top engines of the world. This is quite remarkable, since the evaluation functions of the top engines have all been tuned both manually and automatically over several years, and many of them have been worked on by human grandmasters (Lai, 2015). M. Lai states about Giraffe: "Unlike most chess engines in existence today, Giraffe derives its playing strength not from being able to see very far ahead, but from being able to evaluate tricky positions accurately, and understanding complicated positional concepts that are intuitive to humans, but have been elusive to chess engines for a long time."

Giraffe uses the TD(λ)-Leaf algorithm as a way of generating error signals. It randomly selects 256 positions from the training set and lets the agent play 12 moves. The results of the 12 moves are used to optimise the loss function. The evaluation function is approximated using a 3-layer neural network, consisting of two hidden layers and one output layer. Giraffe, however, does not contain the usage of a replay memory, nor does it contain the addition of a delay between the time of an update to the trainable network and the time the update affects the targets.

To summarise, Baxter, Tridgell & Weaver have focused on creating a chess engine based on Reinforcement Learning. However, features characterising an evaluation function still remains a challenging problem. Since Deep Reinforcement Learning methods have achieved considerable successes in addressing this

issue of feature creation, it is natural to wonder whether Deep Reinforcement Learning applied to chess might produce significant progress to current chess engines. In contrast to Giraffe, DeepChess has been trained both with a replay memory and a delay between the updates of the trainable and target network. The specific methods of creating DeepChess is discussed in the next section, see section 3.

3 Methods

In this section, the literature of Deep Reinforcement Learning is applied to chess specifically. It addresses the theory behind winning King Pawn King endgames, the state representation of chess, the Q-learning network and its hyper-parameters, the addition of an opponent, and finally, the methods of evaluating DeepChess performance.

3.1 King Pawn King positions

Since the whole game of chess is very complex, the endgame has been chosen as a start to apply Deep Reinforcement Learning to chess. The endgame of chess is reached, when most of the pieces are taken. Specifically, the endgame of King and Pawn versus King (KPK) has been chosen to train DeepChess on. In this subsection, the possible positions occurring in this specific endgame are sketched, together with the theory of won and drawn positions.

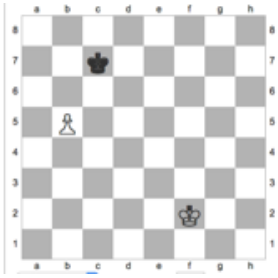
KPK positions are the situations where all pieces are taken, except for one pawn, resulting in positions with two kings and one side having a pawn. Examples are shown in Figure 2.

From whites point of view, KPK positions could be both theoretically won or drawn. Since a position with only two kings is drawn, these positions can never lead to a loss. The strategy for the winning side is to promote the pawn; in other words, to move the pawn up to the 8th rank where it may promote to a queen. It could be said that whenever white is able to promote the pawn, white wins. Winning a game with an extra queen is basic, and therefore, is left out of this project.

The borderline between a theoretically won and a drawn position can be quite narrow, which can be seen in Figure 2. However, these positions can be classified in three difficulty degrees, which have different winning strategies. Firstly, when the pawn is out of reach for the black king, the strategy is to move the pawn up the board before the black king can catch up. Secondly, when the black king is able to take the pawn, the white king should support the pawn in its way to promotion. Thirdly, when the black king is nearly able to keep the white king from supporting its pawn to promotion, the white king should shield

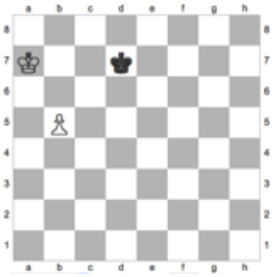
(a)

This position is won, since the black king will not be able to catch up with the pawn (the pawn is out of reach).



This position is drawn, since the black king will take the pawn in a few moves.

(a)



This position is won, since the white king supports its pawn up its way to promotion.

This position is drawn, since the black king is able to keep the white king from moving into a position like Figure 5. This results in a repetition of moves.

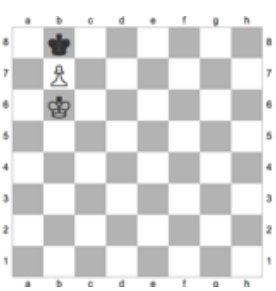
away the black king to enter the second situation.

For training DeepChess, a random position generator has been created, which places all three pieces at a random location on the board.

3.2 State representation

In this section, the state representation is discussed, such that it can be used to apply Reinforcement Learning methods on. It addresses the Markov state, multi-channels and zero-averaging.

For Reinforcement Learning methods to apply, Markov States should be created. A Markov State is a state that is sufficient to formulate a winning strategy only based on the information captured by that state. Chess is a perfect information game, which means no information is hidden to both players. This means that solely the current chess position is enough to tell the best move, resulting in every possible chess position being a Markov state.



This position is drawn, since the black king is in stalemate.

This position is drawn, since the white king is not able to make space for the pawn

(a) Note: Whenever 50 moves have been played without a capture or a pawn move, the game results in a draw too (the 50-move rule).

Figure 2: Different KPK positions. White to move.

To represent a chess position of KPK such that it can be fed to the Q-network, an 8 x 8 x 3 matrix has been created, with numbers corresponding to the location of the pieces. Every piece has its own channel within the representation to avoid stating any implicit ordering between the different pieces, see Table 1. Also, the average of each channel has been made zero, since for some non-linearities, such as a sigmoid function, the gradient is on its steepest around zero, and therefore, the loss function would be optimised faster and would avoid saturated neurons.

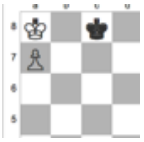


Figure 3: Stalemate position (4 x 4)

[15/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16]
[15/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16]
[15/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16
-1/16	-1/16	-1/16	-1/16]

Table 1: The state representation of Figure 3

The agent would also have to get the number of moves left as an input, would the agent be able to take the 50-move rule into account. Therefore, the 50-move rule is discarded while training, meaning that the game goes on until another terminal state is reached. While training, the game cannot result in a repetition of moves, since the agent will always have a chance of 0.1 of performing a random move. This ensures that the game finishes in a finite number of moves, a requirement of a Markov Decision process[1].

3.3 Network architecture

This section states the architecture of the neural network used to train DeepChess with, followed by the meaning of the output of the neural network.

The network has been made with the use of the libraries Theano and Lasagne. The neural network used to train DeepChess with consists of 5 layers: 1 input layer, 3 hidden layers, and one fully connected output layer. All hidden layers involve a ReLU non-linearity, except for the output layer, which is linear. The architecture of the number of units per layer is pyramided, see Figure 4.

[1]While testing, however, the agent acts greedy to the learned policy, meaning that it never performs a random move. To make up for a repetition of moves, the game is cancelled after 50 moves, resulting in no result (which will count as no-win).

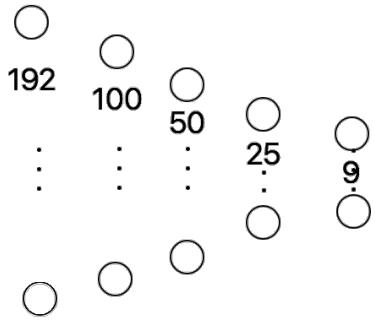


Figure 4: A schematic representation of the trainable network, with the number nodes per layer indicated.

The output layer consists of all possible moves from the agent's side, even moves that result in illegal positions. The possible moves are defined relatively to all piece positions, so the king could move up, or down etc., and not defined as absolute coordinates, since that would make the action space too large. The network outputs a list of Q values that correspond to the value of performing each possible move from the given position.

3.4 Hyper-parameters

This section discusses the hyper-parameters of the epsilon-greedy strategy, the replay memory and the target updates, concerning the Q-learning network.

Size replay memory, N	10.000
Delay update target, C	100 games
Epsilon decay-rate, E	6000 moves
Learning rate, α	0.001
Discount, γ	0.99
Batch size	32
Training games	150.000

Table 2: Hyper-parameters

DeepChess has been trained with an epsilon-greedy strategy, which means that the number of random moves decays over time, while training. When having a simple case, for example, a grid world, where the agent gets a reward when moving into the direction of a particular spot on the grid, epsilon does not have to decrease in order to make the agent learn this task. However, in tasks with only delayed rewards, the epsilon-greedy strategy is necessary for the agent to learn.

All visited states are, together with the chosen actions and the resulting rewards, saved in a replay memory of size 10.000. Every time the agent performs a move, 32 random tuples from the replay memory are picked and used to optimise the loss function. Also, following DeepMind's procedure, discussed in section Deep Reinforcement Learning 2.3, the Q target network gets updated every 100 games, by replacing the old Q target with a clone of the trained network. For a list of all hyper-parameters, see Table 2.

3.5 Reward function

In this section, the rewards of the agent are discussed, together with the handling of illegal moves.

For the agent to learn while playing games, it is necessary that it receives feedback from the environment, which is given in the form of rewards. The agent learns to predict the cumulative future reward for each possible action in each state, by minimising the loss function discussed in Reinforcement Learning, see section 2.1. For reaching a winning state it gets a positive reward, for reaching a drawn state it gets a negative reward (since in white's point of view this is actually losing). If the agent would play the black side too, drawing could be given zero rewards, however, this is outside of the achievements of this project. To encourage DeepChess to find a quick solution, performing a move is also given a small negative reward, see Table 3.

Win	+ 10
Draw	- 10
Illegal move	- 10
Any move	- 1

Table 3: Rewards
Note: there is no reward for breaking the 50-move rule.

To restrict DeepChess from performing illegal moves, the agent is given the same negative reward as drawing when its outputted move is an illegal one. The same state will be fed to the neural network, without any move from both sides. This way the agent learns to avoid illegal moves. While testing, however, the agent should do a move (if not in stalemate), even when the outputted move is illegal. In this case, the agent performs a random move instead. However, this should not happen if the agent has sufficiently learned.

3.6 SunFish

DeepChess has currently only been trained to play the white side, the side with the pawn. To train it properly, a non-random playing opponent is necessary.

This section discusses the specifics of using chess engine SunFish as the opponent.

Python chess engine SunFish has been chosen as the opponent, due to its simplicity and the fact it is written in 111 lines of python code. Whenever DeepChess performs a move, the position is given to SunFish, which returns a move for the black side. SunFish allows the control of the maximal amount of nodes it may search before returning a move, which makes it possible to evaluate DeepChess's performance against different levels of SunFish.

SunFish's representation of a chess position contains a string with dots for empty squares and the first letter of a piece for occupied squares. It is always playing in whites point of view, so the positions needed to be rotated before making it search for a move. SunFish's moves are represented in algebraic chess notation, which had to be encoded to DeepChess's representation.

Unfortunately, SunFish is able to perform illegal moves. Whenever this occurs, the black side performs a random move instead. Fortunately, this happens only when the maximum number of searchable nodes is low, and even when this is low, for example, 200, this happens only in 5% of the games. Also, relative to the maximum number of searchable nodes, SunFish may return no move at all. This is probably a memory issue and has been fixed by rerunning the training loop multiple times.

3.7 Evaluation methods

In this section the evaluation of DeepChess is discussed, together with the theoretical best results achievable and ending with the division of positions into different levels.

In section King Pawn King, see 3.1, we have seen that KPK endgames can result in either a win or a draw. Therefore, it is necessary to separate DeepChess's performance in winning and drawing positions, since DeepChess is only expected to win in a winning position and is not expected to win in a drawn position. Therefore, as an evaluation method, the number of wins in winning positions and the number of wins in drawing positions have been tracked separately.

To know which random starting position theoretically results in a draw or a win, a connection with table base Gaviota was created. Gaviota is a table base including all possible chess endgames with less than 6 pieces, for which each specific endgame's theoretical result is stated.

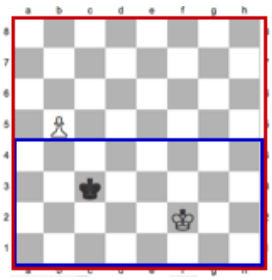
The theoretical best results DeepChess could achieve playing against a perfect opponent, would be to win all winning positions and draw all drawn positions. Playing against SunFish, however, DeepChess could also strive to win in drawn games, since SunFish does not play perfectly, especially when the maxi-

mum number of searchable nodes is low.

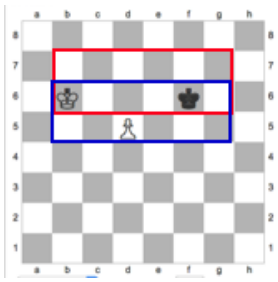
Another intuitive evaluation method is to track the number of moves it takes DeepChess to win. However, in different starting positions DeepChess is aiming for different optimal lengths. Therefore, the amount of moves to win should be tracked per level of starting positions.

Since the difference between the difficulty of starting positions is quite large in KPK endgames, its test environment has been set up with 3 different levels of starting positions: easy, intermediate and hard positions. In easy positions, the white pawn is out of reach for the black king, which means it is only required to move the pawn up the board to win, see the theory of winning such a position in section 3.1 King Pawn King positions. In intermediate positions, the white king is in front of its pawn and the black king next to it. Therefore, it is required to support the pawn with the king. In hard positions, both the white king and the black king are in front of the pawn, which requires knowledge of keeping the black king from controlling the promotion square. In Figure 3-5, boxes are drawn that state where the kings can be randomly placed according to the classification of the different starting positions.

(a)



Easy starting positions



Intermediate starting positions

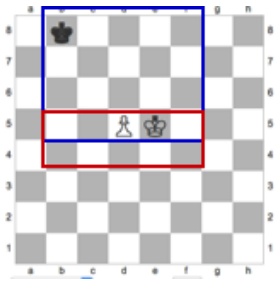


Figure 5: Hard starting positions

Note: The red box indicates the possible positions of the white king, the blue box indicated the possible positions of the black king

As justification for this classification, the number of theoretically won starting positions have been compared to the amount of theoretically drawn starting positions, for each of the different levels. In the easy positions 98 % of the positions are won, in the intermediate positions 74 % of the positions are won and in the hard positions 40 % of the positions are won, see Table 4. Whenever a winning position is more similar to a drawn position, its winning strategy requires more subtle movements, and therefore increases in difficulty. Therefore, the ratio between won and drawn positions serves as a justification for the classification of different levels of starting positions.

-	All	Easy	Intermediate	Hard
KPK endgame	73 %	98 %	74 %	40 %

Table 4: Winning position rates in the different levels

To track the performance during training, the following numbers are plotted: the average loss per move, the highest Q value per state and the amount of wins in winning positions per winning position.

4 Results

To build up the Reinforcement Learning setup, small steps have been conducted before reaching to the final KPK endgame. These are discussed in section 4.1 Experiments. Thereafter, DeepChess has been tested against SunFish, whose results are shown in section 4.2.

4.1 Experiments

4.1.1 The king in a grid world

The first step conducted was to make the agent move the king up to the top of the board, from one specific starting point, with immediate rewards given for every step forward, see Figure 6.

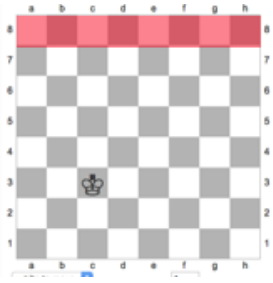


Figure 6: Grid world example with terminal states indicated in red.

Subsequently, the starting position was changed to a randomly placed king, and the agent would only receive a reward for reaching the top of the board. Thereafter, a temporarily fixed opponent king was added, which created a situation where the agent should move the king around the opponent king while walking to the terminal state.

Moves outside the board and moves that would lead into an illegal position were made impossible to perform for the agent. It would choose the best *legal* move from the output of the network. Also, while training, the agent was only exploring, meaning performing random moves.

The neural network consisted of the input layer with one linear layer for all possible output moves.

4.1.2 The addition of the pawn.

The next step conducted was to add the pawn, and make the king and pawn both move to the top of the board, where they would be positioned such that the king would support the pawn to promotion, see figure 7.

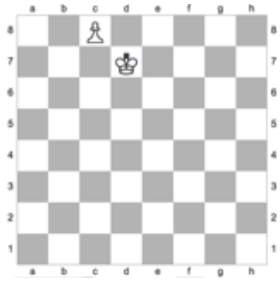


Figure 7: A terminal state with King and Pawn

Firstly, the initial position of the pawn was set at one specific spot on the top of the board, where it would already be in its final destination. Later the pawn was placed randomly on the top of the board, to finally place both the king and pawn randomly on the board. Now, epsilon should decay over time, to give more weight to the greedy policy, like stated in Hyper-parameters, see section 3.4.

When testing, with the agent performing greedy with respect to the learned policy, the agent found itself getting stuck in infinite loops once in a while, mostly at the borders of the board. Until now, it was impossible for the agent to perform illegal moves. This led to the agent sometimes outputting a high Q value for an illegal move - without knowing the move would actually be illegal - which subsequently, would not even be performed. To make the agent learn to avoid illegal moves, a negative reward was given when the agent would *try*

to perform an illegal move, see section 3.5 Reward function . This resulted in a decrease of infinite loops.

To increase its performance, the network was extended to the final network stated in section 3.3 Network architecture.

4.1.3 The addition of a random playing opponent king

A random playing opponent king was added, see figure 8, with the terminal states being the same as in the full KPK endgame. Besides the pawn reaching the top of the board, also draws could occur, namely, the pawn being taken and one of the kings getting stalemated, see section 3.1 King Pawn King positions.

Figure 8: A final KPK endgame situation

The opponent king should only perform a move when the agent has already performed its move, and should always perform a move if it is not in stalemate. To make the learning more stable, a replay memory was added, see section 3.4 Hyper-parameters.

Being trained against a random playing opponent, DeepChess achieved a win-rate of 99 % in winning positions and a win-rate of 86 % in drawn positions, see Table 5.

4.2 Playing against SunFish

Finally, SunFish has been connected to perform the opponent's moves, see section SunFish 3.6. SunFish has been set to play on different levels, corresponding to the maximum number of searchable nodes (maxnodes). DeepChess achieved an overal win-rate of 90 & in winning positions against a SunFish (2000 maxnodes) and an overall win-rate of 88 % in winning positions against a SunFish (10.000 maxnodes), see Table 5 and Figure 9 . The results of a non-learned agent have been added as a baseline.

-	All	Easy	Intermediate	Hard
Random	99 %	99 %	95 %	89 %
SunFish 200	94 %	99 %	76 %	54 %
SunFish 2000	90 %	95 %	67 %	39 %
SunFish 10.000	88 %	92 %	57 %	35 %

Table 5: Winning rates in winning positions

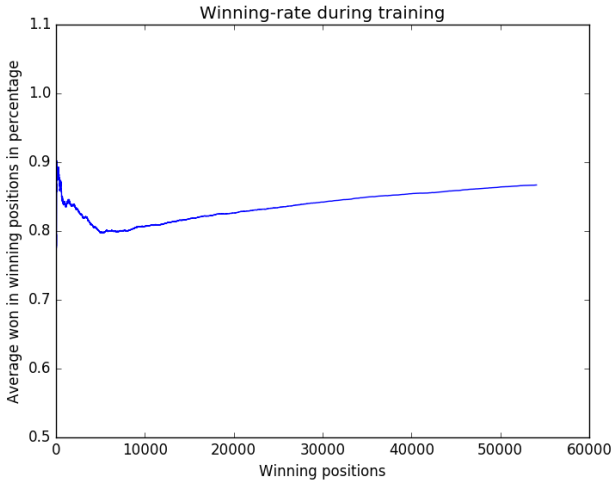


Figure 9: Winning rates every 100 games of training

Noticing the low win-rate in drawn positions, SunFish performs well in keeping the draw when possible, see Table 6.

-	All	Easy	Intermediate	Hard
Random	86 %	90 %	77 %	78 %
SunFish 200	30 %	40 %	19 %	20 %
SunFish 2000	19 %	5 %	10 %	7 %
SunFish 10.000	5 %	5 %	0 %	17 %

Table 6: Winning rates in drawn positions

Also, the amount of moves to win, see Table 7, and the highest Q value outputted by the network per state, see Figure 10, have been tracked.

-	All	Easy	Intermediate	Hard
Random	4.08	3.70	4.84	5.76
SunFish 200	4.14	3.87	5.00	8.35
SunFish 2000	4.01	3.70	4.95	7.41
SunFish 10.000	3.40	3.67	4.66	6.88

Table 7: Moves to win

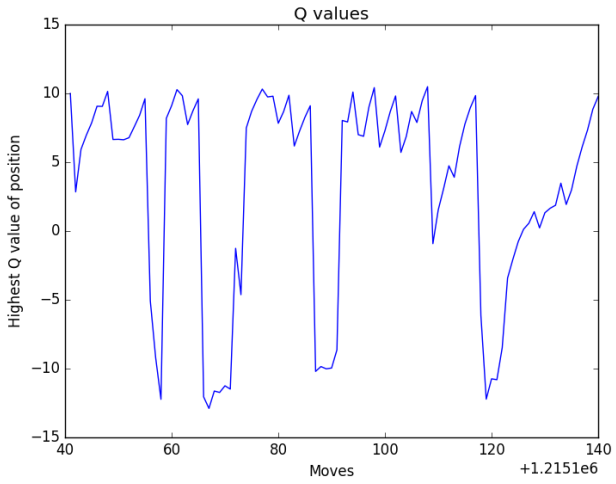


Figure 10: Highest Q values per state

5 Discussion

This section discusses DeepChess's performance, followed by ideas of ways of improving these results. Subsequently, ideas for different opponents are put forward, together with suggestions for future research.

Examining the results in section 4 Results one can notice that DeepChess performs worse in harder positions and against a stronger SunFish. Both of these results fit into the expectations. Also, the number of moves to win corresponds to the degree of difficulty. Looking at the results, it is clear that DeepChess performs well in easy positions, although not 100 %. When a position requires a coordination between king and pawn DeepChess's performance decreases. The concept of opposition, for example, appears to be too subtle for DeepChess to learn.

As an attempt to improve the results, DeepChess's performance in hard positions could be examined, since DeepChess is already able to perform well in

easy positions. DeepChess could be trained with only hard positions, to discover whether the network is actually able to learn these positions well. Experiments with the network architecture and the hyper-parameters could be conducted to ensure the neural network is also able to cope with play hard positions. Also, following DeepMind's procedure of tackling Atari games, convolutional layers could be added. To finally train DeepChess to be able to cope with all possible situations, the distribution of training positions could be generated in such a way that the hard positions appear more often than the easy ones.

When examining the games DeepChess and Sunfish played, it appeared that SunFish did not always perform sensible moves, especially in hard positions. This failure made it easier to win for DeepChess. To make up for this, DeepChess should be tested against a stronger chess engine. Also, to finish the contest between DeepChess and Sunfish, losing positions should be added too. Having a fair match, one could compare their winning rates against each other. Additionally, to be independent of existing chess engines, DeepChess should be trained against itself. The agent would always strive for the maximum amount of reward, and accordingly, would also strive to limit the negative rewards when facing a losing position. Therefore, one and the same agent could play both sides, since it would be striving for maximum achievable rewards in every position.

For future research, I would suggest that DeepChess performance should be increased in such a way that it is able to achieve a 100 % win-rate in winning positions in the KPK endgame. Only then it would be able to compete with the traditional brute-force methods, which are able to solve these endgames easily. Subsequently, DeepChess abilities should be extended to a broader variety of chess endgames, to finally be able to play the full game of chess.

6 Conclusion

With the rise of Deep Reinforcement Learning techniques and the recent successes in the game of Go, chess cannot stay behind in these developments. With respect to current chess engines being mostly based on brute-force methods with help from many grandmasters, I am confident that a new chess engine, independent of any restrictions of human knowledge, could have a good chance of improving on the current state-of-the-art concerning chess engines.

Some attempts have already been realised in applying Reinforcement Learning methods to chess, however, DeepChess is, to my knowledge, the first engine without hand-crafted features that uses DeepMind's findings. DeepChess is based on a 4-layer deep neural network, receiving board positions as input, and has trained against chess engine SunFish. In winning positions it is able to win 88 % of the games against SunFish. However, DeepChess's performance is worse in hard positions, and, to compete with brute-force methods, it should

eventually win 100 % of winning positions.

On the basis of the results of DeepChess, I will provisionally conclude that it is possible to apply Deep Reinforcement Learning to the KPK endgame of chess. As future research, DeepChess should be extended to all chess endgames and eventually the full game of chess, testing against current state-of-the-art chess engines. Only then, it could be confirmed that Deep Reinforcement Learning is a good alternative for the current brute-force methods.

7 References

Baxter, J., Tridgell, A., & Weaver, L. (2001, January). Reinforcement learning and chess. In Machines that learn to play games (pp. 91-116). Nova Science Publishers, Inc..

Bengio, Y. (2009). Learning deep architectures for AI. Foundations and trends® in Machine Learning, 2(1), 1-127.

Block, M., Bader, M., Tapia, E., Ramírez, M., Gunnarsson, K., Cuevas, E., ... Rojas, R. (2008). Using reinforcement learning in chess engines. Research in Computing Science, 35, 31-40.

de Dios, D. R., Cajías, R., & Martínez, V. S. Temporal Difference Learning and TD-Gammon.

Deng, L., & Yu, D. (2014). Deep learning: Methods and applications. Foundations and Trends in Signal Processing, 7(3–4), 197-387.

Hall, M. A. (1999). Correlation-based feature selection for machine learning (Doctoral dissertation, The University of Waikato).

Hsu, F. H. (2002). Behind Deep Blue: Building the computer that defeated the world chess champion. Princeton University Press.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

Lai, M. (2015). Giraffe: Using Deep Reinforcement Learning to Play Chess. arXiv preprint arXiv:1509.01549.

Lison, P. (1996). An introduction to machine learning.

Michalski, R. S., Carbonell, J. G., & Mitchell, T. M. (Eds.). (2013). Machine learning: An artificial intelligence approach. Springer Science Business Media.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... Petersen, S. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533.

Silman, J. (2006). Silman's Complete Endgame Course: From Beginner to Master. Urban Media Comics.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

Silver, D. Reinforcement learning course - Lecture 3: Planning by dynamic programming

Sutton, R. S., & Barto, A. G. (1998). Reinforcement learning: An introduction. MIT press.

Tsoukas, H. (2005). Do we really understand tacit knowledge?. Managing Knowledge: An Essential Reader, 107.