Learn to Hex

Kalle Kromann & Bjørn C. V. Bennetsen
Supervisor: Oswin Krause
Bachelor project at Datalogisk Institut - Københavns Universitet

June 2019

Abstract

In this project we will implement a simulator of the board game Hex and use it to train an agent to play the game of Hex using Reinforcement Learning (RL) through self-play. In the project we theoretically examine and implement two different RL algorithms; CSA-ES which is an evolutionary strategy method and TD(0) which is a prediction learning method. Broadening our focus to these two significantly different RL methods gives a broader view of the field, but it also means we have a greater attack vector on the problem which is to train an agent to play Hex. Our implementation of the simulator and RL methods are both written in C++, and we are using the Shark machine learning library for our learning algorithms.

To evaluate our results we conduct several experiments, where we look at the win-rate of the agents against both a random strategy and previous versions of the agent as they are being trained. We are focusing on these results because they give us an indication of whether or not the agents are actually learning anything. The results seems to show a general trend across each experiment, that they are in fact improving and are winning consistently against the random strategy. None of our trained agents however showcase particularly intelligent behaviour and are performing worse than the average human beginner-level, indicating further research and work could be done to improve on the results.

Contents

1 Introduction

In our thesis we are going to implement a simulator of the board game Hex and use it to train an agent using reinforcement learning methods in self-play, to play the game. We will be using a machine learning library for C++ called `Shark` which is developed at DIKU and for this reason we will also write the simulator in C++.
We are going to look at two different reinforcement learning methods, prediction learning with temporal differences also known as TD and evolution strategies or ES for short. We have chosen these two methods because they are both popular methods in reinforcement learning and are significantly different in the ideas behind them and how they are implemented. The specific algorithms we use for experimenting are $TD(0)$ which is a special-case of $TD(\lambda)$ and CSA-ES which is a step-size adapting evolution algorithm widely used with the popular CMA-ES algorithm.
Hex is the popular name of a board game that was invented and first introduced by the Danish mathematician, writer and poet Piet Hein in 1942, at the Niels Bohr Institute. Interestingly, it was later independently reinvented by the American mathematician John Nash in 1948 and is also known as Con-tac-tix, Nash, or Polygon. Hex is special type of board-game that belongs to the category of "Connection Games", where the players are trying to complete a specific type of connection. In Hex, a game cannot end in a draw which has been proved using the *Brouwer Fixed-Point Theorem* [1]. Another interesting fact is that John Nash has proved that for games played on symmetrical boards, there exists a winning strategy for the player starting the game. However, this does not tell us what the actual winning strategy is. Using game-tree search-techniques, complete winning strategies has been found for board-sizes up to 7×7[1], but for larger game-boards a general winning strategy has yet to be found. This to us, makes Hex an interesting game to experiment with. Even though we probably aren't going to find an optimal solution, even for small board-sizes, we might find, that it uses some interesting strategy in order to win the games.

Our purpose with the project is in part to create an efficient simulator and implement the learning algorithms correctly to obtain a result which indicates whether or not these algorithms are sufficient to train an agent to play Hex. But part of our purpose is also to get acquainted in the machine learning field of science in which we have had very limited formal education as of yet. Machine learning has gained a lot of popularity in the recent years and is applicable in countless scenarios so we want to have a deeper understanding of the field than just the overall concepts, and see if it is something that we find interesting and would like to focus on in the future. We will also use the project to improve our knowledge of C++ which neither of us have used to an extensive degree before. C++ is a language packed with features and libraries compared for example to C and this also allows for well constructed and easily understandable code which is still very efficient. But it also means there is a lot to learn and a lot of ways to over-complicate solutions, so working on a project like this which requires a decent amount of C++ code will hopefully teach us some valuable knowledge and lessons.
In our report we will first in section 2 explain the relevant background theory that is used in our implementation and especially go into detail with the specific reinforcement learning methods we use. This section also details the game mechanics of Hex. Section 3 describes our Hex simulator, how it is implemented and how it can be used through a basic interface of functions. Section 4 describes our implementation of the two learning algorithms, how our program can be used to train an agent with either of the methods and how we specifically are conducting the experiments. The sections 5 and 6 presents and discusses the results of the experiments and our project. Finally in section 7 we conclude our project. The source code for our implementation is attached with our thesis and can be used to reproduce our experiments, which is explained in the appendix.

[1] `https://webdocs.cs.ualberta.ca/~hayward/hex7trees/`

2 Background

2.1 Reinforcement Learning

In reinforcement learning the learner only needs some way to sense the state of its environment and a way to take actions that affect the state, to be able to learn how to maximize a given reward signal. The learner must through many episodes of trial-and-error try different actions and observe how they effect the reward signal, even though the reward is not always immediately known after each action. In pure reinforcement learning the learner starts out without any previous knowledge about its environment, such as what it means to take an action, and in which situations a reward is received. Take for example the environment of the game chess where there are very specific rules for which moves are valid for each chess piece. Here the learner must try moving the pieces around by taking random actions to discover which moves are actually valid, and then also which moves lead to preferable situations where the learning agent can end up winning games consistently. In order to have a better starting point however, it can be useful to have some knowledge of the environment be known to the learner, like what actions are possible to take which we do in our implementation.
Reinforcement learning can also be used in combination with other forms of machine learning, which has been done successfully by several research teams like in DeepMinds AlphaGo [2] and even more recently with their StarCraft II agent [3] where they first use supervised learning by training a policy with a data-set of expert moves and then use reinforcement learning to maximize the reward of self-play of the policy. In this project we will look at two different methods, temporal difference learning and evolution strategies, to train an agent with pure reinforcement learning self-play.

To explain how reinforcement learning works, and how it can be solved, we will as in Chapter 3 of Suttons book [4] try to frame the problem in the context of a Finite Decision Markov Process or MDP.
The MDP is modelled by having an *agent* and an *environment*. The agent is the learner that is trying to learn a goal by interacting with the environment.
The agent interacts with the environment in discrete time-steps from $t_0, t_1, ..., t_m$, where it observes the state of the environment S_t at time t, and then depending on S_t it chooses an action A_t. Taking the action changes the environment state from S_t to S_{t+1} which is observed by the agent at the next step $t + 1$. Then on the basis of the new state and the action the agent took, it receives a reward signal R_{t+1}.
For a finite MDP the set of states, action and rewards is finite, and so a sequence of agent-interactions with the environment would produce a finite sequence of states, actions and rewards:

$$S_0, A_0, R_0, S_1, A_1, R_1, \ldots, S_t, A_t, R_t$$

With the problem framed like this, we can say that the purpose for the agent/learner is to maximize the reward it receives by interacting with the environment. That does not mean choosing actions that increases the immediate reward, instead we want to choose action so that the future rewards is maximized. To be exact, we want the cumulative future reward-signals received to be maximized, and achieve this by choosing the right actions.
The intention of the reward signal is to articulate what specific behaviour that we want to encourage. For example in a board-game, we obviously want to maximize the number of wins we experience. So in order to obtain winning-behaviour, we would at least have to give a reward-signal when the agent wins the game. But one can achieve winning behaviour in many ways. In some games we often need to maximize something, e.g points, distance etc, in order to win a game, and in these cases the reward-signal might be given, for example, when the agent scores a lot of points. It is also possible to give a negative reward in case the agent performs bad.

Moving on, we can then describe the *expected return* as G_t at time-step t. Where G_t is the sum of future rewards from state t to the final state T. This can more formally be stated as:

$$G_t = \sum_{i=0}^{T} R_{t+i} \tag{1}$$

There are other ways of defining the expected reward, where each reward is discounted so that older rewards weighs less in the sum of rewards, but since we don't want to discount the rewards, we'll use (1).

With this in place, we introduce the concepts of *value*-functions and *policies*.
The value-function is a function that attempts to give a value for the particular state the agent is in. The value indicates what the expected return is. The expected return is however still obtained as a result of the actions the agent takes. Therefore the value $v(s_t)$ function is defined in relation to some policy π. A policy π is formally a mapping from states to probabilities of taking each action. It defines the decision-making process of taking actions. A policy could for example just be choosing random actions.
We can express the value-function v for a state s following the policy π as $v_\pi(s)$, which is the expected reward following the policy π:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{T} R_{t+i} | \; S_t = s \right] \tag{2}$$

Just as we can have an state-value function we can have an action-value function which is defined as the value of taking action a in state s. So we can define action-value function $q(s, a)$ under the policy π as:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{i=0}^{T} R_{t+i} | \; S_t = s, A_t = a \right] \tag{3}$$

A policy is just the way that the agent takes an action, and the job is to find a policy which maximizes the expected reward. So we can more formally say that a policy π is better or equal to some other policy π' if the expected reward is greater, which is only when $v_\pi(s) \geq v_{\pi'}(s)$ for all s.
When this is the case, π is an *optimal policy* and is denoted $\pi*$, and the optimal policy has an *optimal state-value function*(and action-value function) denoted v_* and is defined as $v_*(s) = \max_\pi v_\pi(s)$ So the solution to problem of reinforcement learning can now be seen as a problem of finding a policy π which maximizes the expected reward. However as we will show in section 4, in some problems we only need to find/estimate the state-value function in order to find the optimal policy. That is we can have a constant policy of which we find an optimal value function, but as we'll also see we can also just find an optimal policy, which in turn will give us an optimal action-value function.
However, here an optimal solution is defined only in finite MDP's, and even so, if the state-space is very large it would take an infeasible amount of time and computing power to achieve such an optimal solution. Therefore one can only approximate the optimal solution when the state-space is too large. This is often the case in board-games with large a state-space, for example in Hex.

2.2 Prediction Learning with Temporal Differences

Temporal Differences is a class of learning algorithms that all solve the same problem, that is learning to predict future behaviour using past experience.
Conventional prediction learning was initially about training a model to predict certain

things after training it against training data, i.e. with supervised learning. The general difference between MC-methods and TD-methods, is that MC-methods will have to wait until the end of an episode in order to learn anything. They have to observe all states, actions and rewards that happens during an episode, and then use this experience to predict the future. This is not necessary when using TD-methods, as it can perform the learning-step in each step of an episode. In our setting however, we will use TD with the exact same setting as in MC because the problem can be modelled in a finite MDP.
We will consider the problem of multi-step prediction as it is also the type of prediction problem we have in Hex. That is where the observations during an episode is on the form: $x_1, x_2, x_3, ...x_T, z$, where x_t is a vector of real-valued observations, which in the game of Hex could be the encoded game-state. z is a real-valued number denoting the actual outcome of the sequence, that in Hex could be the result of the game.
For each observation made during the episode, a prediction P_t is made that is the estimated/predicted value for z at time t. The prediction P_t is often obtained using a function that is parametrized by a set of modifiable weights, let's call them w.
To re-frame this in a setting of RL, we define the observations x_t as the states s_t, which is the states that the agent sees. The learner chooses actions A_t according to the policy π. The reward here is given after each interaction with the environment. In the setting of Hex, the reward signal is only given at last turn of game, i.e. when one player has won. So in a sequence of t states, the actions and rewards looks like this:

$$S_0, A_0, 0, S_0, A_0, 0, \ldots, S_t, A_t, 1$$

We'll define the predictions P at the observation x_t, $P(x_t)$ as the state-value function v at state s_t, $v(s_t)$. The state-value function is, like P, also parametrized with a set of weights. The process of learning to predict becomes a task of modifying the weights of $v(s_t)$, so that we maximize future rewards. The weights can in a more(or less) simple case just be updated at the end of an episode, by changing the weights by the sum of the increments seen at each step of the episode:

$$w \leftarrow w + \sum_{t=1}^{m} \Delta w_t \tag{4}$$

In a supervised-learning approach we would express the observations as an episode of observations and outcomes. The learning that happens here, depends typically on some error between the predicted value $v(s_t)$ and the actual outcome z. And so a supervised-learning update step would look like this:

$$\Delta w_t = \alpha(z - v(s_t)) \nabla_w v(s_t) \tag{5}$$

Where α is the learning rate, typically between 0 and 1. $\nabla_w v(s_t)$ is the gradient of state-value function $v(s_t)$ with respect to the parameters w. Almost any machine learning task can be formulated in terms of minimizing a cost function or error. It can be seen that the update-step in (5) is the derivative of the square-loss error $(z - v(s_t))^2$:

$$\alpha \Delta_w \frac{1}{2}(z - v(s_t))^2 = \alpha(z - v(s_t)) \nabla_w v(s_t) \tag{6}$$

We want to minimize this error by modifying the weights:

$$\min_w \sum_t 1/2(z_t - v(s_t))^2 \tag{7}$$

The most simple function to represent $v(s_t)$ with, is with a linear function of the form $v(s_t) = w^T s_t$. Here the gradient $\nabla_w v(s_t)$ is just the feature vector/state s_t. Another case of TD is when the state-value function $v(s_t)$ is computed by a non-linear Artificial Neural

Network which is dependant on weights w and the state s_t. Here the update rule is the same as in (5), but the difference lies in how to compute the gradient $\nabla_w v(s_t)$.
Equation (5), however depends on the final outcome of an episode, so we would have to wait until all observations of an episode has been observed. So that actually means one can't use equation (5) in an incremental manner, but as mentioned this is not a problem for us because we actually need to wait until the outcome of the episode is determined. However we want to show that we can use the predictions made during an episode to learn.
The idea is to represent the error between the prediction and the actual outcome $z - v(s_t)$ as the sum of changes in the predictions encountered during an episode. That is:

$$z - v(s_t) = \sum_{k=t}^{m} (v(s_{k+1}) - v(s_k)) \quad \text{where } v(s_{m+1}) = z \tag{8}$$

The above is of course only true if we assume that $v(s_t) = v_\pi(s_t)$, i.e. that we have the optimal state-value function.
If we combine (4) and (5), and insert (8), we can derive the following:

$$\begin{aligned} w &= w + \sum_{t=1}^{m} \alpha(z - v(s_t)) \Delta_w v(s_t) \\ &= w + \sum_{t=1}^{m} \alpha \sum_{k=t}^{m} (v(s_{k+1}) - v(s_k)) \Delta_w v(s_t) \\ &= w + \sum_{k=1}^{m} \alpha \sum_{t=1}^{k} (v(s_{k+1}) - v(s_k)) \Delta_w v(s_t) \\ &= w + \sum_{t=1}^{m} \alpha (V(s_{t+1}) - v(s_t)) \sum_{k=1}^{t} \Delta_w v(s_t) \end{aligned}$$

Then at time t we can describe the weight increment as:

$$\Delta w_t = \alpha (v(s_{t+1}) - v(s_t)) \sum_{k=1}^{t} \nabla_w v(s_k) \tag{9}$$

So now the weight increment can be computed just by having the prediction error between two successive steps and the gradient $\Delta_w v(s_t)$ at state s_t.
With this in place, we now have a way of representing the error $z - v(s_t)$ as the sum of changes in predictions over the experienced episodes. And so we get, for each difference in the predictions $v(s_{t+1}) - v(s_t)$, a value Δw_t which tells us how the weights should be increased/decreased in order to make the future predictions better. More exactly, if we made a sequence of predictions $v(s_{t+1}) - v(s_t)$ for a sequence of states s_t, and we were using equation (5), we would expect the predictions of future states of which it has predicted something earlier to be different, because it uses the sum of all changes experienced.
Connecting this again with the square-loss error function we can describe the weight-update after obtaining the outcome z as:

$$w = w - \alpha \frac{d}{dw} 1/2 (z_t - v(s_t))^2 \tag{10}$$

This leads us to the last class of TD-methods called TD(λ), where we will see that (5) is just a special case of the TD(λ)-method.
As stated, in the case of (9), the predictions made during an episode are equally altered due to the sum $\sum_{k=1}^{t} \nabla_w P_k$. One could however imagine scenarios where one does not want to

alter the predictions in an even manner, but instead give greater alterations to prediction made in the near past. This gives birth to λ, where in an exponential-manner the past k predictions are used to increment the weights according to λ^k where $0 \leq \lambda \leq 1$. And so we could write the update step as:

$$\Delta w_t = \alpha(v(s_{t+1}) - v(s_t)) \sum_{k=1}^{t} \lambda^{t-k} \nabla_w v(s_k) \tag{11}$$

Here we can see that when $\lambda = 1$ we get equation (9) where all past prediction are altered in an equal manner with the sum of past changes in predictions. This version is called TD(1). The most simple form of the TD(λ) is when $\lambda = 0$. Here the weight update step only uses the difference between the current and the most recent prediction, so the update step becomes:

$$\Delta w_t = \alpha(v(s_{t+1}) - v(s_t)) \nabla_w v(s_k) \tag{12}$$

When the state-value function is for example a neural-network, it requires the gradient to be computed in a special way using a method called *Backpropagation*. This method we will not go into detail explaining, but we will use method however, indirectly by the way shark computes the derivative of such a neural-network, to back-propagate the error in the neural-network.
To connect with the square-loss error, the weight-update step can also be described with the following gradient-descend update step:

$$w = w - \alpha \Delta_w 1/2(z_t - v(s_t))^2 \tag{13}$$

We will in section 4 show how we have implemented the TD(0) to train an agent, via. self-play to learn the game of Hex. We try to approximate the optimal state-value function.

2.3 Evolution Strategies

Evolution strategies are machine learning methods inspired by the theory of evolution by Darwin. The common idea for evolution strategies is that there exists a generation of individuals which are all candidate solutions for the problem. Each individual is then tested to see how well they solve the problem and receives a fitness score according to how well they performed. Then the individuals with the highest fitness score are selected for the next generation, imitating natural selection and survival of the fittest. When creating a new generation it is common to employ some form of genetic recombination. This could mean taking two "parent" individuals and creating a "child" by using half of each parents features, but can also be a more general recombination of all the selected individuals. A new generation is created by sampling from a probability distribution, which is often a Gaussian distribution,

$$p(x|\mu_t, \sigma_t^2 C_t) \ , \tag{14}$$

where μ_t is the mean, σ_t the standard deviation and C_t the covariance matrix at generation t. Ranking the individuals and finding a mean for the next probability sample can be viewed as a way of selecting the fittest individuals from a generation. The idea is that each generation will have a mean fitness score that is a little better than the previous generation, so the algorithm should converge towards an optimum. Random mutation is also modelled in this scenario since we are sampling from a distribution, completing the analogy to the theory of evolution.
This kind of policy search is called direct policy search. The policy space is searched for an optimal policy by sampling from a parameter space which maps to policies in the policy

space. The probability distribution p mentioned before is therefore sometimes called the search distribution.
In the algorithm we use, the search distribution is assumed to be isotropic which is a simplification that means $C_t = I_d$ and we shall use this simplification in the following also. This algorithm is authored by Oswin Krause and is explained in great detail in his article [5]. At each step of an ES algorithm a generation is sampled and the mean is updated according to the fitness of the generation using a learning rate α_t. Following the Information-Geometric-Flow (IGO) theory[6], an iteration of the algorithm can be described as optimizing the objective function,

$$\max_{\mu_t, \sigma_t} \mathbb{E}_{p(x|\mu_t, \sigma_t^2 I_d)}\{w_t(f(x))\} \tag{15}$$

where $\mathbb{E}$ is the expected value of a sample and w_t is a weighting function that weighs each point value, such that a large weight means that the probability of sampling a better point is small. Weighing the points is analogous to the recombination of features from the fittest individuals of the parent generation as when updating the mean using the weights, the next generation will move more towards the higher weighted points.
Sampling i points for the next generation is done like so,

$$x_{i,t} = \mu_t + \sigma_t z_{i,t},\ z_{i,t} \sim \mathcal{N}(0, I_d)\ , \tag{16}$$

and each sampled point is evaluated on the weighted objective function to obtain the weights $w_{i,t} = w_t(f(x_{i,t}))$. It is worth noting that for the normal distribution sampling it holds that $\mu_t + \sigma_t \mathcal{N}(0, I_d)$ is the same distribution as $\mathcal{N}(\mu_t, \sigma_t^2 I_d)$.
The mean is now updated like in the following,

$$\mu_{t+1} = \mu_t + \alpha_t \sum_i^{\lambda} w_{i,t}(x_{i,t} - \mu_t) \tag{17}$$

where α_t is the learning rate at time t.
A problem with simple evolution algorithms with a fixed σ, also called step-size, is that they either get stuck at a local optimum if the step size is too low and so doesn't explore enough, or if it is too high it might never find an optimum because it is too quick to abandon a search point. Likewise, a low learning rate will result in a very slow convergence and a too high learning rate could mean that the search points would jump back and forth and never settle on something good. The step-size and the learning rate is therefore often adapted during training and can be seen as a way of zeroing in on an optimal solution. A method of adapting the step-size is the 'Cumulative Step Adaptation' method, referred to as CSA. This method is often used in CMA-ES algorithms, since adapting just the covariance matrix will not affect the overall scale of the distribution but rather morph the distribution such that it is not linear in all directions. In the following, we will describe the CSA algorithm as introduced by Hansen[7] and Krause[5].
CSA directly adapts the step-size σ based on the correlation between steps using a so-called evolution path which is defined as a sum of consecutive steps where the step-size is disregarded. For example, an evolution path of the generation mean could be constructed like this,

$$\frac{\mu_{t+1} - \mu_t}{\sigma_t} + \frac{\mu_t - \mu_{t-1}}{\sigma_{t-1}} + \frac{\mu_{t-1} - \mu_{t-2}}{\sigma_{t-2}}$$

In practice the following equation is used for constructing the evolution path,

$$p_{t+1} = (1 - c_{p,t})p_t + \sqrt{c_\sigma(1 - c_{p,t})\mu_{eff,t}}\frac{\mu_{t+1} - \mu_t}{\sigma_t} \tag{18}$$

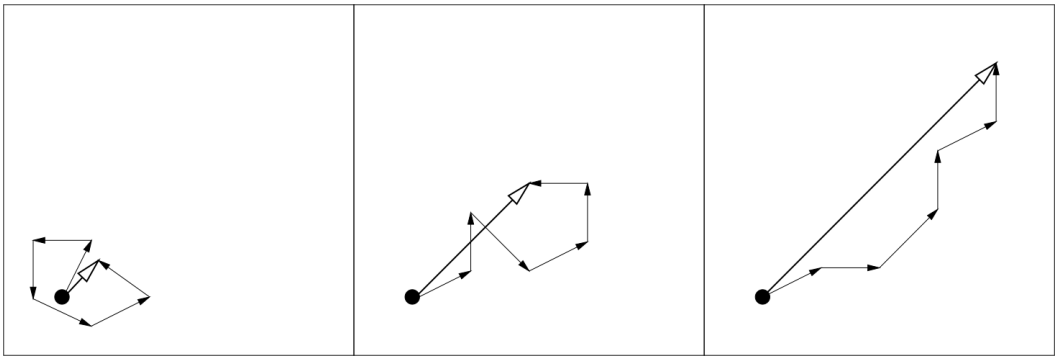


Figure 1: Shows the evolution paths from 6 steps in 3 different situations. The evolution paths are the arrows with a white arrowhead, and the black arrows are individual steps. The black circle is the starting point and as can be seen the evolution paths go straight from the starting point to the point the last step points to, which is because it is the cumulative sum of those steps.

In this equation $\mu_{eff,t}$ is the effective sample size and $c_{p,t}$ is a learning rate where $1/c_{p,t}$ is the backward time horizon of the evolution path, i.e. how many consecutive steps that are taken into consideration. Adapting the step-size is done by comparing the length of the path with expected length in an unbiased random selection scenario where consecutive steps are uncorrelated. Uncorrelated steps are the desired situation and indicates that the strategy is nearing a minimum. We will borrow a figure, here referenced as figure 1, from Hansen's paper [7] that visualizes the different situations of step correlation. If the actual evolution path is longer than the expected path it means that the steps are going in similar directions so we can say they are correlated. This is the situation seen in the rightmost square of figure 1. Since these steps are going in the same direction, fewer steps with a greater step-size could have been taken to get to the same point, so in this case the step-size is increased. On the other hand, if the actual path is shorter it means the steps are going in different directions and are cancelling each other out, as they are in the leftmost square of figure 1. In this situation the step-size should be decreased as that would allow for more precise steps to near the minimum that the strategy is stepping around. In the ideal situation the steps are more or less perpendicular to each other as the situation seen in the middle square of figure 1, because this suggest an unbiased selection of thus uncorrelated steps. Adapting the step-size by comparing the lengths of the actual path $\|p_{t+1}\|$ and expected evolution path $\mathbb{E}\|\mathcal{N}(0, I_d)\|$ is done in the following update step,

$$\sigma_{t+1}^2 = \sigma_t^2 \exp\left(\frac{c_{\sigma,t}}{d_{\sigma,t}}\left(\frac{\|p_{t+1}\|}{\mathbb{E}\|\mathcal{N}(0, I_d)\|} - 1\right)\right) \tag{19}$$

where $d_{\sigma,t}$ is a dampening factor and is according to Hansen's paper [7] based on in-depth investigations of the algorithm.
The algorithm we use in the following is a noise-resilient adaptation of CSA by Krause [5], which adapts the learning rates based on the estimated noise-level of the function. See the referenced article for a full explanation.

2.4 Self-play

In a competitive game like chess or hex there are several agents which interact with the state. The philosophy of self-play reinforcement learning is that the learning agent can never be better than it's hardest opponent. So in order to properly explore the problem space and become a master player that can potentially defeat world champions, as with AlphaGo, the learner must take the role of each agent in so called self-play. If the learner is

only taking one agents role, the learning will be limited to the level of play that the other agents have. And if the other agents used the same strategy again and again, the learner would rarely discover new situations that it could learn from. Because the learning agent is playing both sides of the game, the hope is that it will push itself further and further as it becomes a better and better player.
Theoretically that would mean there is no limit for how good a player the agent can become. Practically however we run into the problem of local minima/maxima, which is a prevalent problem in machine learning algorithms. We're trying to solve the problem,

$$\arg\min_{w} f(w), w \in \mathbb{R}^d \tag{20}$$

which is finding the minimum of the black-box function $f : \mathbb{R}^d \to \mathbb{R}$ obtained at $w^* \in \mathbb{R}^d$. A black-box function is a function that takes some input and gives some output, but where the actual implementation isn't really relevant for us. Instead of understanding the structure of the function, we use a neural network to try to approximate this black-box function. The black-box function is in our case a function that given a board state can pick the move that maximizes the players chances of winning (the actual implementations of this varies a bit between the two algorithms we're using). The function is effectively supposed to take the best possible move given some board state. When thinking about this function, it is hard to imagine there being any "global minimum", a point where the function always finds the best move. Because how do we define the best move in a competitive game? Would it not depend on the opponent being played? That being said, it can be proven that there exists a winning strategy for the starting player of Hex, which is based on the fact that there must always be one winner. The important point right now is that even though this strategy might exist, it is far from trivial to figure it out and then follow it.
It is hard to imagine a global minimum, but very easy to imagine a lot of local minima. A local minimum could be a strategy that often leads to wins, but is vulnerable to some counter-move by the opponent. As stated, it is not hard to imagine a lot of these local minima, and this could be a problem in self-play reinforcement learning. If the learning agent finds one of these minima and is only playing against itself, it might level out at both players winning 50% of the time, meaning there is no real motivation for further exploration. Each time the agent tries something new it has a chance of affecting the current minimal strategy badly (especially in a game like Hex) and so it would move back to the strategy that tends to lead to a good amount of games won. In order to try to circumvent getting stuck in an early insufficient local minima one can try using a better starting point for the self-play algorithm. This is what DeepMind achieved with the preliminary supervised learning of AlphaGo, where the agent started by learning from expert plays. This results in a starting point for the self-play algorithm where the strategy is already pretty sophisticated, using well known tactics and moves. The learning of an agent with a good starting point will also eventually get stuck at some local minimum, but it has avoided a lot of pitfalls from the early learning stages. As mentioned, in this thesis we're using pure reinforcement learning with a more or less totally random starting point, so we're not expecting our resulting agents to perform well against real players.
The implementation of the self-play part varies between TD and ES. This is because of the inherent differences in the methods which we will explain further in section 4. From the "viewpoint" of both learning algorithms however, it is irrelevant that they're in fact used in self-play. The self-play part can be seen as a wrapper around the actual learning part of the algorithm. Playing a game to evaluate the performance of an agent and then using that to optimize the black-box function can be done in other ways, like playing against a random move-picking strategy or an expert.

2.5 Shark

Shark [8], is an open-source, fast and feature-rich machine learning library written in C++. The long list of features in shark, makes it a suitable library for almost any type of machine-learning task. It comes with a lot of built-in functionality, e.g. for building models such as linear models, but also more complex, such as Artificial Neural Networks.
The following sections draw heavily on the information stated at the Shark website and the tutorials provided there.

Models

A model can generally be seen as the solution to some problem.
In machine learning, we often want to find a model which predicts or responds with the best possible answer, given some input. Models in Shark can be seen as a function taking some input and returning an output. Models are often parametrized in some way, with internal parameters, so that the task of learning a model becomes a task of finding the right parameters that makes the model behave in (some) optimal way.
All models in shark are derived from the templated-class `AbstractModel<InputType, OutputType>`. For example, a linear model can be constructed with an input-shape and an output-shape with the templated constructor:

```
template<class InputType=RealVector, class ActivationFunction=LinearNeuron>
shark::LinearModel< InputType, ActivationFunction >::LinearModel(
                                                    Shape const & inputs,
                                                    Shape const & outputs = 1,
                                                    bool offset = false)
```

Here `Shape` is a class for holding the shape of the data, typically for vector data. It allows for specifying the dimensions of the data that is to pass through the model.
Models inherit the private member function `eval` from the it's base-class `AbstractModel`, `void eval (InputType const &input, VectorType &output)const`, which takes some input, evaluates the model, and stores the output in the shape that is defined by the model. The linear model we have presented only describes a single point of input and output. What makes shark fast is the way it optimizes tasks, where models can take a whole batch of inputs and output a batch of outputs. This allows some algorithms to exhibit more efficient/better learning, since they are not restricted to look at one point at the time.
An `AbstractModel` also implements some additional high-level interfaces that implements parametrization and serialization of the models. The parameters are handled by the `IParameterizable`-class and the serialization by the `ISerializable`-class, which provides the `AbstractModel` with several methods like `inputShape` and `outputShape` that gets the model input- and output-shape. Additionally it provides functionality for writing and loading parametrized models, so that they can be saved and used later.
Shark also allows for creating non-linear models, such as Neural Networks.
A neural network can be implemented by stacking layers of, e.g., linear models, by specifying an associated neuron-activation function, which can be used to make the model non-linear. Considering a linear model f applied with an activation function g is described as $y = f(x) = g(Ax + b)$.
Shark has an implementation of a `ConcatenatedModel` which provides the `>>`-operator for stacking layers together.
So for example, a network with 3 layers, an input layer, a hidden layer and an output layer could be created like this:

```
LinearModel<RealVector, RectifierNeuron> inputLayer(input_inShape, hidden_inShape);
LinearModel<RealVector, RectifierNeuron> hiddenLayer(hidden_inShape, out_inShape);
LinearModel<RealVector> outLayer(out_inShape, out_outShape);
ConcatenatedModel network = inputLayer >> hiddenLayer >> outLayer;
```

We will in section 4 show what specific models we have worked with.

Objective Functions
An objective function is some function that formalizes some objective, that are to be optimized in some way. In Shark they can be used together with optimizers in order to solve optimization-problems.
An objective function in shark can be implemented as a derivation of the base-class `AbstractObjectiveFunction<SearchPointType, ResultT>`. Here the type `SearchPointType` is the type of search-point the objective function uses. This could for example be the parameters of some model that the objective function are to evaluate. The type `ResultT` specifies the return-type of the objective function, which for single-objectives is a `double`. The objective function also allows for specifying if the underlying model has derivatives which can be used, for example by optimizers.
Shark provides several objective functions, for example the `ErrorFunction` which is used widely in supervised learning. Here the objective is to minimize some error between the model-predictions and some training data.

Optimizers
Optimizers are used to optimize the value of some objective-function. In Shark optimizers are iterative functions, that for each step tries to find some search point $x(t+1)$ at time $t+1$ so that $f(x(t+1)) < f(x(t))$. Shark distinguishes between two different types of optimizers. As shortly described above, objective functions can make the derivatives available(if they are available by the model) so gradient-based optimization algorithms can be used.
An other type of optimizer uses a derivative-free- or direct approach to finding the right search-points. A simple case could just be to linearly search trough points in some defined space. Or as we use in this thesis, an evolutionary-inspired algorithm which searches the point-space more purposefully.

2.6 Mechanics of Hex

Hex is a strategic board game for two players played on a hexagonal grid. The traditional shape of the game board is a 11×11 rhombus, but the board can be of any shape and size. Players alternate placing stones/bricks on unoccupied tiles in the grid. The objective is then for the players to form a unbroken chain of stones from one side of the grid, to the other. The first player to make such a connection wins the game.
In Hex the players have "perfect information", which means that when a player is making a decision, she is perfectly informed of every event that has previously occurred in the game. Hex is also completely deterministic, after the starting player is selected there is no random behaviour like die rolls.
Because there is an advantage in making the first move of a game, as we'll discuss later the game is often played using the *swap-rule*, where the second player has the option to take the move that the first player just made, or she can just continue to play as player two by placing a stone somewhere else. This helps helps balance the game, so that player 1 does not exclusively have an advantage in starting. Also, it encourages player 1 not to play the best starting position, which on odd-number boards often is the center-position. We did not implement the swap rule in this project because we think it would complicate our results.
Figure 2 shows an example game played on a 5×5 board.

3 Hex Simulator

This section describes our implementation of a simulator of the Hex game in C++. This simulator is what we use for our experiments to create a Hex AI using reinforcement learning. The reason for implementing the simulator in C++ is that it can be seamlessly integrated with the Shark library [8] that we use for implementing the reinforcement learning.

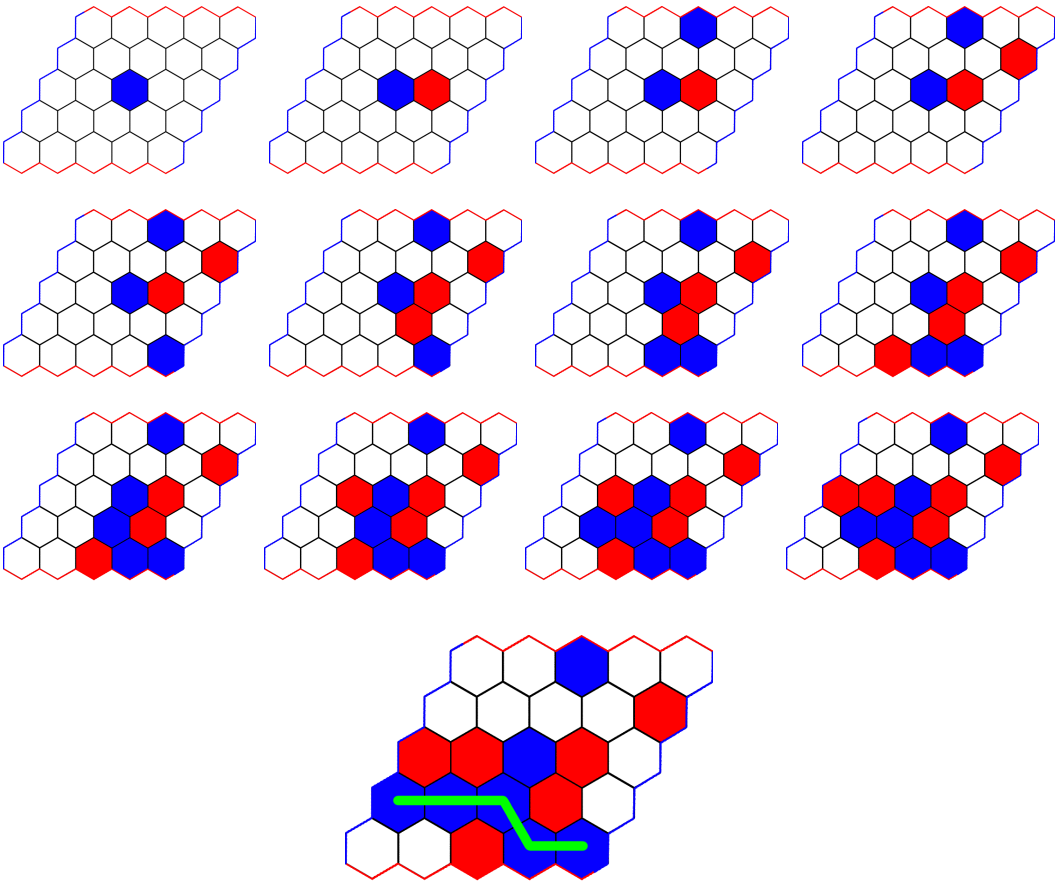
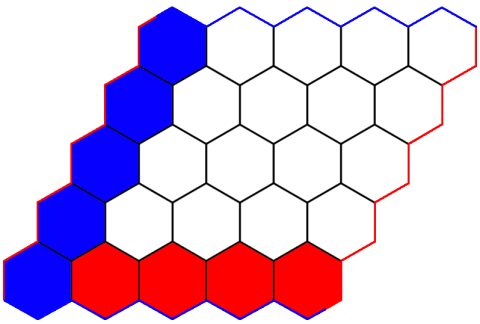


Figure 2: Example game on 5×5-board, where player 1 wins after 13 turns.

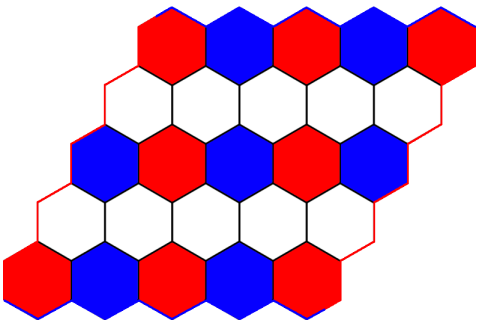
3.1 Data Representations and Memory Management

C++ requires manual memory management, and since we want to train a neural network using 10.000+ training runs we need to be careful to avoid memory leaks as this would gradually slow down the simulator. There are 2 main memory usages in the simulator, the game board and linesegments. The game board consists of `Tile` objects, and the board and tiles are allocated on the program stack at the start of the program, and reused by resetting each tile after a game. Stack allocated memory is safe in the sense that it can not lead to memory leaks, since it will be live from start of the program and until the program ends.

A `Tile` object can be part of a `Linesegment` which is a class representing a group of connected tiles of the same color. The `Tile` class has a direct or indirect reference to the linesegment it is a part of, but the linesegments does not store the tiles that is a part of itself which spares some memory and time. The purpose of the linesegments is explained in the section about checking for winning condition. A game of Hex can be played from start to end with only 2 linesegments being used, if both players always place tiles next to their previously placed tiles, as shown in figure 3a. But in another game where the players place tiles that are not next to any of their previous placed tiles until it is impossible to find another of such positions, there might be many linesegments. For example on a standard 11x11 rhombus shaped board there might be up to $33 * 2 = 66$ linesegments, before it is impossible to find a position that does not have any neighbours of the same color. This



(a) Game state with 2 `Linesegments` where the player who started also won.



(b) Game state with 15 `Linesegments` which is the worst case for a 5x5 board. This game is not finished, but the amount of `Linesegments` will drop after this state.

Figure 3: This figure shows how the amount of `Linesegments` used depends on how the game is played.

situation is shown in figure 3b where any next move will be next to a neighbour of the same color, regardless of whose turn it is. After this point the amount of linesegments starts to shrink, since they will be merged with each other.
We decided to dynamically allocate memory for linesegments using the smart pointer `shared_ptr`[2]. A `shared_ptr` is responsible for freeing the memory it uses, and this is done by keeping a count of references to the pointer. As soon as the last object that has a reference to the `shared_ptr` is either destroyed, or the reference is removed (perhaps by setting it to another `shared_ptr`) the memory is freed. This simplifies our code since we don't have to manually free pointers when merging linesegments or when the program ends, and it also ensures the correctness of our memory management so we have no memory leaks.

3.2 Game Interface and Basic Mechanics

A game can be instantiated like so (assuming the `Hex.hpp` library is properly included),

```
Hex::Game game;
```

After instantiating a game the method `takeStrategyTurn` can be called in order to advance the game using a given strategy, or `takeTurn` can be called directly with an action. The `takeStrategyTurn()` method is called with two 'strategies', one for each player. Depending on which players turn it is, the corresponding strategy is used to choose an action. After choosing an action, `takeTurn` is called with that action. This function executes the action and gives the turn to the next player. If the game is still going afterwards the functions return true, and if the game ended after the previous action they return false. If the game is over, the winner is the player that is not the current active player, since the turn was just handed over. Using the method `getRank()` we can query the rank of a player. The winner has rank 0 (first place) and the looser rank 1. The method `asciiState()` returns a string containing the current game-state, which can be printed to the terminal, like figure 4 shows. This interface allows for a simple way of simulating a game and checking who won,

```
1 while (game.takeStrategyTurn({& strategy0, &strategy1 })) { // Simulate game
2     std::cout << asciiState() << std::endl; // Print game state to stdout
3 }
4 double r = game.getRank(0); // Rank of  player 1
```

[2] `https://en.cppreference.com/w/cpp/memory/shared_ptr`

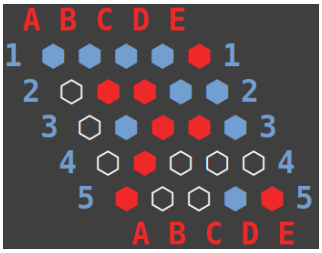


Figure 4: Example of how it looks when printing the state returned by asciiState() to a terminal.

After a game is over the method `reset` should be called to clear the game board and other variables that are specific to a game. `takeTurn()` can also be accessed directly as it is a public member which can be useful if you have a specific action to take.

3.3 Strategy

As mentioned, the method `Game::takeStrategyTurn()` uses the strategy of the active player to choose a move. The `Strategy` class is a base class that different strategies can inherit from. The class contains 3 virtual methods that can be overwritten, `numParameters()` which gets the number of parameters (relevant for strategies that use neural networks), `setParameters()` that sets the parameters (again relevant for neural networks). The final method `getMoveAction()` is the only method used by `Game::takeTurn()`, the other methods are used for controlling the neural network of a strategy, if the strategy uses one. `getMoveAction()` takes the current game board and returns a vector with probabilities for placing a tile on each possible position of the board.
We have implemented several strategies that inherit from the `Strategy` class, and so can be used to play a game of Hex. Games can be played by two players using the same type of strategy or two different strategies.

- `CSANetworkStrategy` is our strategy for use in ES training experiments. This strategy uses a neural network with an input layer that is fed by the gamefield, to generate the probabilities. More about the neural network and how this strategy works will be further explained later.

- `TDNetworkStrategy` is our strategy for use in TD training experiments. This strategy is quite different from the rest because of how we have implemented the TD algorithm, so the strategy must be used with `takeTurn` instead of `takeStrategyTurn`. More about this will also be further explained later.

- `RandomStrategy` just returns equal probabilities for each tile on the board, which means a random empty tile is picked. We use this strategy to verify that the algorithm is actually learning something, in which case the win-rate of the network strategy should go towards 100%, as it should be trivial to beat a random playing player in Hex unless he's really lucky (or you're really bad).

- `HumanStrategy` is a strategy that uses human input to play. That means it will block and prompt input from the user and take the requested move. We use this strategy for playing against the AI so we can test it's strength and see how well it plays against us. After getting a valid input from the user the strategy returns probabilities of minus infinity (approx.) for every tile except the one the player chose which is 1. That means it is actually the tile that the user picked which is placed.

3.4 Checking For Win Condition

A game of Hex is won if one of the players have a connected path between both of her assigned sides of the board. We don't care about finding the shortest path, any connected path will do.
A naive and expensive approach to checking for a winning condition would be to start at the newly placed tile, and then going through each neighbour or until the other side of the board is reached. This approach would not only be pretty expensive, and depend on the board size etc., but it would require a quite complex algorithm, which could be difficult to design and implement.
Instead we employ the before mentioned `Linesegment` class, and group each tile by being members of a `Linesegment`. When placing a tile we just have to update the line segment to tell whether the game was won by placing that tile.
The `Linesegment` class is very simple, and just stores two booleans, `Connected_A` and `Connected_B`, and two functions, one for merging with another line segment, and one for merging with a newly added tile. Merging just refers to OR'ing the booleans of the line segments (or newly added tile).

```
bool Merge(bool other_A, bool other_B) {
    Connected_A |= other_A;
    Connected_B |= other_B;
    return Connected_A && Connected_B;
}
```

The `Tile` class has two properties that are used to define which line segment it is a part of,

- `m_linesegment`: This is a shared pointer to a `Linesegment` object or nullptr.
- `m_tile_ref`: This is a pointer to another tile that is part of the same line segment or nullptr.

The properties are mutually exclusive, so if one is defined the other is a null pointer. When we want to find out which line segment a tile belongs to we use the following method,

```
std::shared_ptr<LineSegment> GetLineSegment() {
    if (m_linesegment != nullptr) {
        return m_linesegment;
    } else if (m_tile_ref != nullptr) {
        return m_tile_ref->GetLineSegment();
    }
}
```

An analogy for how this works is that we ask the tile, "What is your line segment?". If the tile has the reference it will give it to us, if not it will respond with, "Ask this other tile that is also part of my segment". We then ask the same question to the referenced tile until we get the line segment reference.
The critical part now is to make sure that that each tile has either a valid line- or tile reference, and that there is no cycles in the references.
When a new tile is placed, we check if it has any neighbours. If there are no neighbours a new line segment is created for that tile. If the tile was placed on one of the players own board edges, one of the booleans of the new line segment is set to true. If the new tile has one or more same colored neighbours we add the new tile to the line segment of the first found neighbour by setting the tile reference to point at that neighbour. If the new tile is on an edge, the corresponding line segment boolean (A or B) is set to true. For each other neighbour of the new tile that does not already share the line segment, we merge their line segments. This is done by calling a recursive function `ReferLineSegment()` which is another member of `Tile`.

```
void ReferLineSegment(Tile* newref) {
    if (m_linesegment != nullptr) {
        m_linesegment = nullptr;
        m_tile_ref = newref;
    } else if (m_tile_ref != nullptr) {
        m_tile_ref->ReferLineSegment(newref);
    }
}
```

This function will call itself recursively on the tiles tile reference, until it finds a tile that has a reference to a line segment. This reference is then removed, and instead the tile reference is set to point at the passed parameter, `newref`. So it is the last found new line segment that exists after a tile is placed, and the other line segments are all merged into it. Figure 5 shows an example of two line segments being merged. In that example there are two line segments at the beginning of a turn, and a tile is placed that touches both segments, so they are merged into one. As can be seen in the example, the line segment L2 is found first, so L2 is merged into L1 and then freed, and we end up only having L1.
After looping through each of the 6 neighbours, and merging any line segments, we know instantly if the game was won or not. If the final merged line segment has both booleans set to true, the segment has a path from one side to the other and the game is won.

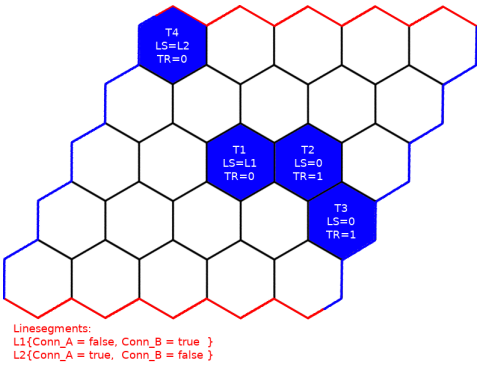
4 Implementation

In this section we will describe the non-trivial parts of the implementations of our two algorithms which is the basis for our experiments, and explain how the program we have written can be used to experiment with training a Hex-playing agent.

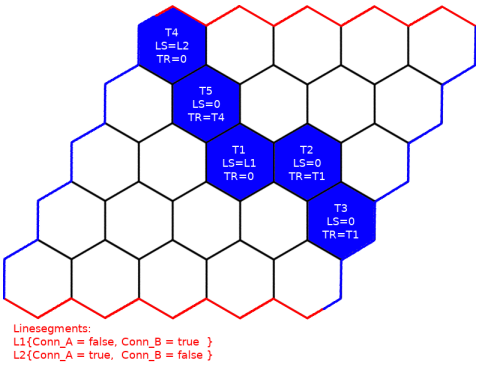
4.1 Game State Representation and Self-Play

We want to train one agent to play the game of Hex through reinforcement learning obtained via self-play. As described earlier, the game of Hex needs two players in order to play the game, but because of the symmetry of the game, we can train an agent which can learn- and play as both players. Because the objective of player 2 is just the 90-degree-clockwise rotated objective of player 1, we can say that player 2 has the same objective as player 1 if the board is rotated 90-degrees in the counter-clockwise direction, so that the view it is like that of player 1, which is trying to connect a path between the horizontal opposing sides of the board. The idea is that by doing this, we will have a simpler objective to learn, which is the same objective for both players.
Because we want to train the agent using the machine learning library Shark, this also forces some restrictions on how the algorithms are implemented.
As described, the agent we are training is only seeing the game-state as seen from the perspective of player 1. This means in practice that when the agent is playing as player 1, the state at time t is just s_t, which means that the state of the game-board is just the original rotated game-board. After the agent has taken a turn as player 1, the next state of the game is s_{t+1} and can be described with the function $transition(s_t, a)$ which gives the state after taking action a. Let's call this transition o_t, so we have $o_t = transition(s_t, a)$. When the agent is playing as player 2, the state the agent is seeing here is the rotated view of the state o_t, because this would rotate the view of player 2 into the view of player 1. Let's call this state $flip(o_t)$ where $flip$ is a function that just rotates the state into the view of the other player.
So during a game of t steps, the states that the agent sees is the following sequence of states:

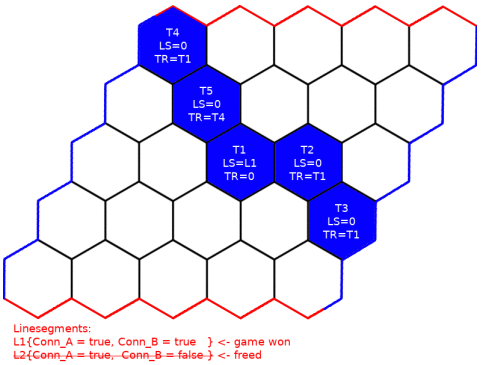
$$s_0, flip(o_0), s_1, flip(o_1), \ldots, s_t, flip(o_t)$$



(a) Before the tile T5 is placed, there are two line segments, L1 and L2. L1 is a segment of 3 tiles, and L2 is a segment of 1 tile.

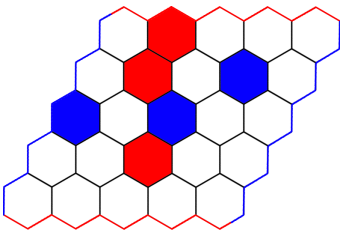


(b) After placing T5, the first neighbour it finds is T4 (because it searches from top-left). T5s tile reference is thus set to T4, and T5s line segment reference is null.

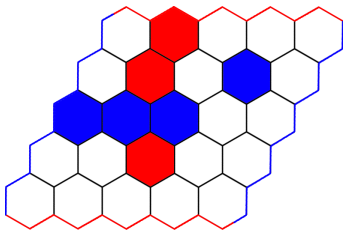


(c) The next (and last) neighbour it finds is T1, so `ReferenceLineSegment(T1)` is called on T5, and since T5 have a tile reference, the function is called again on T4. This tile has a line segment reference, so that is set to null, and its tile reference is set to T1 (from the `ReferenceLineSegment` argument). Now there's only one line segment, and this linesegment is connected to both of blues sides, so blue wins the game. Also shown is the fact that every reference to L2 is lost, so the smart pointer is freed.

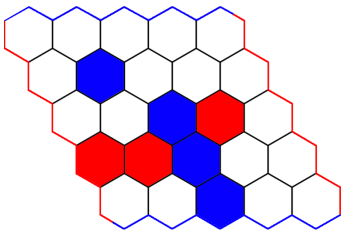
Figure 5: Example of two line segments being merged after a tile (T5) is placed that connects them.



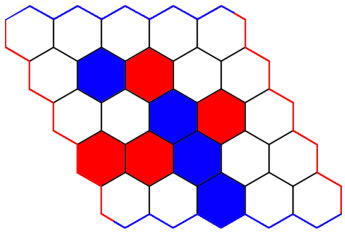
(a) State s_0 where blue is the next player to make a move.



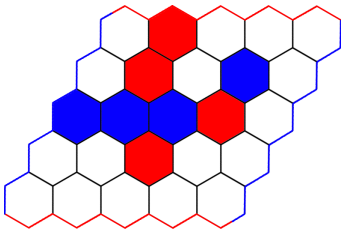
(b) State o_0 obtained after blue taking an action and changing the environment by $transition(s_0, a)$.



(c) State $flip(o_0)$ where red is now the horizontal player. This is what the red player observes.



(d) State $flip(s_1)$ obtained after red making a move on the rotated board.



(e) State s_1 after the board has been rotated back. Now it is blue who is the horizontal player again, about to take a new action.

Figure 6: Example of a Markov chain of state representations that changes after the players take actions. A state-transition is made between (a) and (b), and between (c) and (d). The changes from (b) to (c) and (d) to (e) are state-transformations where the state is rotated. The states actually observed by the agent are state s_0 in (a), $flip(o_0)$ in (c) and s_1 in (e)

Figure 6 illustrates the above by showing what the state representations look like after taking actions as each player.
With this set up, we can move on to describe how we have implemented the algorithms.

4.2 TD(0)

In this section we will explain the theory behind- and how we have implemented TD(0) for training an agent to play the game of Hex. Our implementation is inspired by Tesauro's implementation of TD(λ) in his famous TD-Gammon paper [9] . However we do not implement TD(λ) but TD(0).
Temporal-difference learning solves the problem of learning to predict. In the context of learning to play the game of Hex, we want the agent to learn how to predict the outcome of a game of Hex, and then use these predictions as a "policy" for how to take actions in the game, i.e. take actions which leads to best or worst values, depending on the perspective.
The way we solve the prediction problem is by approximating the optimal state-value function $v_\pi(s_t)$. Our policy π is already defined,and because of this, the problem we need to solve is not finding a policy π, but instead to approximate the optimal state-value function $v_\pi(s_t)$ for our given policy π.
We define the weight $\boldsymbol{w}$ parametrized state-value function as $v(s_t, \boldsymbol{w}) = f(s_t, \boldsymbol{w})$, where $f(s_t, \boldsymbol{w})$ is some function parametrized by the weights $\boldsymbol{w}$, which maps from states to probabilities - but let's just denote it $f(s_t)$ and $v(s_t)$.

The update step can then be written as:

$$\Delta w_t = \alpha(v(s_{t+1}) - v(s_t))\nabla_w v(s_t) = \alpha(f(s_{t+1}) - f(s_t))\nabla_w f(s_t) \tag{21}$$

With this definition of the state-value function in place, we'll explain the reasoning behind taking actions with the action-value function. The action-value function is defined as $q(s_t, a)$ and is the value of a state s_t if taking action a, or rather what the value(expected reward) of taking a certain action a in state s_t. What we want the agent to do is to choose actions that leaves the opponent with the worst state-values, or the lowest probability of winning, which in turn (should)give the current player, the agent is controlling, a higher probability of winning.
The agent takes the action $a*$ that minimizes the value of the next state, which is the turn of player 2. When the agent plays as player 1, this means that it takes actions according to:

$$a* = \arg\min_a q(flip(transition(s_t, a)), a) = 1 - f(flip(transition(s_t, a)))$$

That is, the agent chooses the action $a*$ which minimizes $q(flip(transition(s_t, a)), a)$ and $1 - f(flip(transition(s_t, a)))$, which is the value of the next state i.e. for player 2 or one minus the value of the player 1's state.
In almost the same way, when the agent is choosing actions as player 2, the actions are chosen according to:

$$a* = \arg\min_a q(flip(o_t), a) = 1 - f(flip(transition(flip(o_t), a)))$$

Here we remember that $o_t = transition(s_t, a)$, so when the agent is playing as player 2, it also picks the actions that minimizes the value of the next state, which is the state of player 1.
Now we have defined the state-value function and how we use it to let our agent take actions. We will however never end up with an agent that takes good actions, if it never learns from the predictions/values it makes.
Even though the TD-methods are well-suited for incremental learning, where the weights $\boldsymbol{w}$ can be updated in each step of an episode, we do not do this in our case. This is partly because we have a very simple reward-signal that is only given at the last step, can't learn something that isn't rewarded, and because we can calculate the gradient of $v(s_t)$ in a efficient way with the Shark-library.
As discussed we have defined $v(s_t) = f(s_t)$, where $f(s_t)$ is the model/function that we use as our state-value function. In Shark, the `Model`-interface let's us construct a lot of different models and to compute the derivatives thereof. Because Shark aims at maximum computational speed, the derivatives is computed as a weighted sum of the models partial derivatives. This means that we only need the model inputs- and outputs, together with some weights to compute the model-derivative. During an episode we store the model inputs, i.e. the states s_t or $flip(o_t)$ and the model-output values i.e. $v(s_t)$ or $v(flip(o_t))$. The weights are used in calculating the derivatives, and is defined as the TD-errors $\delta = R + V' - V$, where R is the vector of rewards storing all zeros except the last element, V is the vector of values storing $V = \{v(s_1), v(flip(o_1), \ldots, v(s_t), v(flip(o_t))\}$ and V' is storing $\{v(flip(o_1)), v(s_2), \ldots, v(flip(o_t)), v(s_{t+1})\}$ seen during a t-step episode. So δ is a t long vector holding the TD-errors seen during an episode.

The model $f(s_t) = v(s_t)$ is implemented in the `TDNetworkStrategy`-class which implements several things regarding the model. That is for example flipping the board into the view of the other player or for evaluating the model given some input. It also has the functionality which implements the action-decision making discussed.
The algorithm itself is implemented in the `TDAlgorithm`-class. When initialized, it has

the algorithm-specific parameters, such as the learning rate α and the weights used in the model.
The training is, as described, done in episodes where the agent changes between taking moves for the players until the game is won by either player. During the episodes we save the state(which is the encoded game-board) and the state-value and use them to compute the TD-errors. These saved states and values are also required by Shark to compute an internal state of the model, using the input and output values, which is then used to compute the derivatives. The last two lines of an episode is then computing the derivatives using the TD-errors as the weights, and then updating the weights with the derivative and the learning rate.

```
RealVector derivative;
m_strategy.GetMoveModel().weightedParameterDerivative(
    stateBatch, valueBatch,
    tdErrors, *state, derivative );
// update weights
m_weights += m_learning_rate*derivative;
```

4.3 CSA-ES

The implementation of our evolution strategy algorithm for training an agent to play Hex is based on our supervisor Oswin Krause's published algorithm which is a variant of CSA-ES that is more noise-resistant than the original. We were also supplied with his code for the learning part of the algorithm which was made for a similar board game simulator. So for this method we have only implemented the interface between our Hex simulator and the learning part, and the neural network structure that is being trained to value the board states. This was a good start for our own learning process, because we had a part of the final algorithm we knew was correct so we could focus on getting the rest to work. When we got this algorithm to work with our simulator we could move on to implementing our own TD learning algorithm with a lot more ground knowledge of both Shark and machine learning in general. It also means we can experiment with two different methods of machine learning and compare both the implementations and the the results of training. We have already discussed the algorithm in the background section, so this section will focus on how we use it to train an agent to play Hex through self-play.
In this implementation, the search distribution is used to sample from a parameter space containing parameters for a neural network. The neural network is used when playing a game to evaluate a board state, and this evaluation is used to pick the best move according to the network. So the chromosomes of the individuals in a population are parameter sets for the neural network. What we want to achieve is a parameter set for the neural network such that when it evaluates a board state, it outputs precise probabilities for winning if the current player places a tile on any of the board positions. The network does not "understand" this of course, so all we can do is to interpret the output as these probabilities and then try different parameter sets until we have a set that lets the network output make sense to interpret like this. We do of course have control over the size of the output which we set to be equal to the total amounts of tiles on the board, i.e. the board size squared, so there is a value for each tile in the output. In Hex you can't place a tile on top of another players tile, so the values corresponding to each non-empty tile are set to have a probability of 0 and the rest are normalized so they are between 0 and 1. A value is then sampled uniformly such that the tile with the highest possibility of leading to a win is picked with the highest probability, but there is a diminishing chance of picking a lower value. This procedure defines the strategy of the individuals, and when referring to different strategies we mean different sets of parameters for the neural network that is used in the strategy.
In ES we sample a population based on the previous generation, and each new individual

is a candidate player with a slightly different strategy. In order of generating a new better population for the next step through self-play we need to find the individuals with a strategy that performs the best on average against the other individuals, i.e. the fittest individuals of the generation. To find the best performing strategies we play them against each other and rank them based on these games. Even though our Hex simulator is fast, it would still be costly to have each individual play against the rest as that would have a exponential time complexity. Instead we obtain an unbiased estimate of their fitness by playing each individual against two other random individuals from the generation. For example on a population size of 20 we play $20 * 2 = 40$ total games instead of $20^2 = 400$ which is a speed up effect of factor 10, and that effect will increase with the population size.
After estimating the fitness of the individuals, the recombination weights and the length of the evolution path are computed and these are used to update the mean and the step-size. Then an offspring population is generated from these variables by sampling from a Gaussian distribution.

4.4 Training an Agent

After compiling our source code an executable called `hex` is created which can be used to train an agent using either TD or ES. The executable can be run with the command line arguments '`td`' to train an agent using TD or '`es`' to train one using ES.
Training an agent refers to taking a step in a learning algorithm and updating the parameters for the neural network. We will refer to a parameter set for a neural network as a model for the strategy.
We have defined two helper classes for training an agent which initializes what is needed for the specific method and contains some member methods for advancing training, and evaluating the current state of the agent. These methods are,

- `step` - Takes a step of the TD or ES algorithm and keeps count of steps taken.
- `playExampleGame` - Plays an example game with the current state of the agent as both players and prints each intermediate state to the terminal. This gives an insight into how the agent is playing and if it is learning any good strategies. It also shows that both players are going for their respective objective (connecting horizontal or vertical opposing sides) which indicates that our state representations are correct. Remember, both players see themselves as the horizontal player, but the actions are flipped for the vertical player. In example games TD is not epsilon greedy and ES uses the mean of the population, such that we should get the currently highest performing strategy.
- `playAgainstRandom` - Plays a game where player 1 is the current state of the agent being trained as in the above (the network strategy), but player 2 is played by a random strategy which places tiles totally at random. Only the final state is printed to the terminal, because here we're most interested in the win/loss ratio against a random strategy. After a random game the win statistics are updated and displayed to the terminal. If our training algorithms are actually working we should see a pretty even win/loss ratio at the beginning, but after some time the ratio should start to go in player 1's favour and approach a 100% win rate for the network strategy. This main purpose of this method is so we can know if the agent is actually learning anything or is just continuing to play at random as in the beginning.
- `playAgainstModel` - Takes the name of a saved model and plays a game between the current agent strategy and a strategy that loads the saved model. We can use this method to see if the agent is actually improving and winning against previous versions of itself. After some preliminary training with a more or less total random agent, the agent should start consistently winning against previous versions. There

may be outliers to this if the agent is exploring heavily which means the strategy is not very stable. When the learning of an agent is getting stuck at local optimum, the win-rate of the newer model is expected to go towards 50%. If the win-rate gets below 50%, something is probably wrong and the agent might be unlearning a "good" strategy.

The trainers also have a method for saving a model `saveModel` and one for loading a model `loadModel`. As mentioned, a model is a set of parameters for a neural network, so saving a model means writing a file with these parameters. We save the model using built-in functionality of the `Shark` library which saves the file as a boost serialized archive. The two functions take the name of the file to save or load. If the training process is somehow interrupted, like the computer shutting down unexpectedly, the training progress which could have been several hours in the making would be lost. Therefore it is a good idea to save the model intermittently during training, because then most of the progress could be restored by loading the saved model before starting training again. This feature also allows for playing the current model against a previously saved model as described above, and for playing a model against a human player.
While training an agent it can be interesting to play against it yourself, because you can test directly how well the agent plays, like if it tries to block your moves and if it goes for a direct winning strategy or not. For this purpose we have made a strategy which can be used with `takeStrategyTurn` that prompts the user for where to place the tile. Playing against a trained agent can be done simultaneously with training by starting a new hex process in the terminal. The type of the model and which saved model to play against are supplied a command line argument,

```
$ ./hex esplay someModel # play against ES trained model "someModel"
$ ./hex tdplay someModel # play against TD trained model "someModel"
```

Input for taking a move in the terminal is on the form [A-Z][1-9][1-9]?, e.g. 'C11' which means column 'C' and row 11.
Since writing out each move in the terminal is slow and prone to errors we created a python3 application which can be used to play against trained agents using a mouse in a GUI window. The application uses TkInter[3] for creating the GUI and opens a sub-process which runs the `hex` executable and communicates with the process using pipes. Figure 7 is a screenshot from the application where a 5x5 game has just been won by the blue player which is played by the trained TD model. During a game the tiles of the board can be clicked in order to make moves. There is also a restart button which restarts the game by restarting the hex process with the selected model, and a button next to the model name which opens a file dialogue that can be used to select the model to load. The board and window size changes according to the board size used in the hex executable. The python application needs to know if it should run a TD or ES version of the hex play process so this is given at the command-line when starting the application. Here a model can also be specified instead of using the file dialogue. The user must take care to load correct models, that means models that were trained with the same method (i.e. TD or ES) as the hex process is run as and which were trained on the same size board as is built in the current hex executable. If an incorrect model is loaded the app will freeze and should be killed, as we didn't get around to handling this. Following shows an example of how to run the application from a command-line,

```
$ python3 playhex.py --algorithm=td --model=build/TDmodel.model
```

[3] `https://wiki.python.org/moin/TkInter`

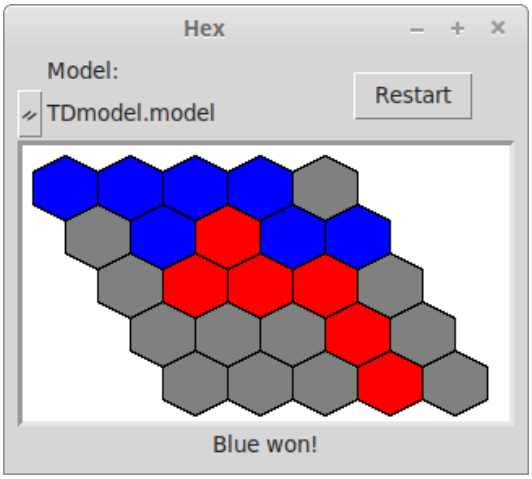


Figure 7: Screenshot from python application `playHex.py`

4.5 Experiments

In this section we will reason about the experiments we will conduct. The experiments we conduct will be focused around showcasing the performance of the agent at different times during the learning process. We will quantify the performance in two different ways: how well the model plays against a random-playing opponent using `playAgainstRandom` and how well it performs against earlier versions of the model using `playAgainstModel` where at each 100 steps, the current model is saved and plays against the previously saved model. For simplicity we will only be using neural networks with one hidden layer as our models. The starting player for the experiments is selected at random so the agent is not biased to play as the starting player.
In some experiments we will vary different parameters like the board size, the amount of neurons in the model and the activation functions used. Then we have an experiment where we look at several sessions of the same setup and find the average and another experiment that is run for far longer than the rest. We have tried to include a lot of experiments to get a broad view of the training quality, but as an effect of this we will not comment extensively on every single result.
In most experiments 50.000 episodes were run because some of the experiments take a long time to run, and using the same amount of episodes makes the plots easier to compare. The results are gathered as 500 points with 100 steps in between them. For each point we play 100 games, either against a random opponent or a previous model depending on the experiment, and the points which make up the blue graphs are the average win-rate in those 100 games. That is, the specific models average win-rate. We also collect the total amount of won games out of all previously played games which we use to find the average win-rate of the agent across the entire session. This serves to smooth out the win-rates and gives a better view at the overall progress of the learning agent. This total average win-rate is shown in all of our plots as the yellow graph.
As a baseline for our results we have gathered the win-rate between two random players over 50.000 games. Figure 8 shows the results from the baseline experiment which is the average win-rate of player 1. The x-axis which shows episodes is only in the range of 0-500, but at each point 100 games are played giving a total of 50.000 games, so it is comparable to our experiments. Since the starting player is picked at random we would expect the win-rate to be close to 50% in unbiased play. In the first couple of thousand games the average win-rate was relatively unstable, but nearing the end it was very close to 50%. The data for this plot was generated from a 5x5 board size, but the results are similar on other

board sizes.

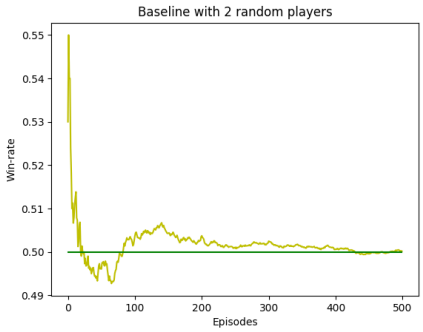


Figure 8: Baseline plot showing the average win-rate of player 1 in 50.000 games where both players pick unbiased random tiles. Here the blue graph is not shown and the green graph is just showing where 50% is.

5 Results

In this section we will explain the setup of the experiments we have conducted, and present their results. Each experiment was conducted using both the TD(0) and the CSA-ES algorithm. The results will be discussed in section 6.

5.1 Board-Size Variation

In this experiment we varied the board-size of the Hex game, between 3x3, 5x5 and 7x7. The neural-network we used was using the `RectifierNeuron`-activation function in the input- and hidden layer. The neural network was had a fixed number of neurons in the hidden layer with the input-output shape $80 - 40$.
Figures 9, 10 and 11 show the plots generated from training sessions with the TD(0) algorithm on the 3 different board sizes, while figures 12, 13 and 14 are from CSA-ES training sessions.

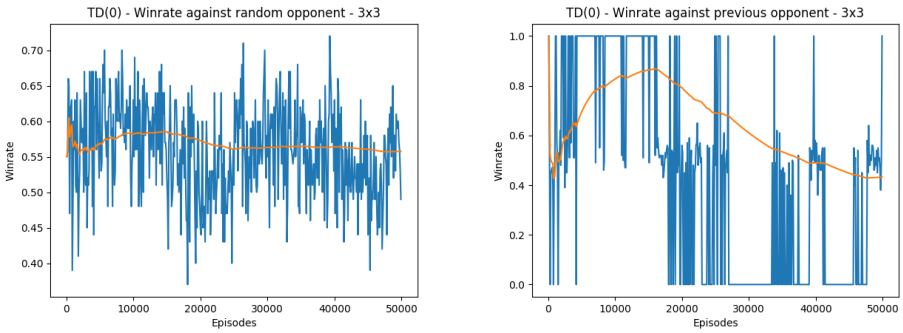


Figure 9: Results from 50.000 episodes of the TD(0) algorithm on a 3x3 board.

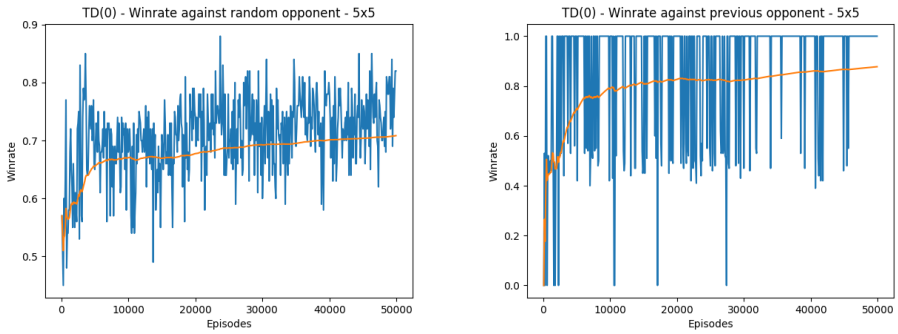


Figure 10: Results from 50.000 episodes of the TD(0) algorithm on a 5x5 board.

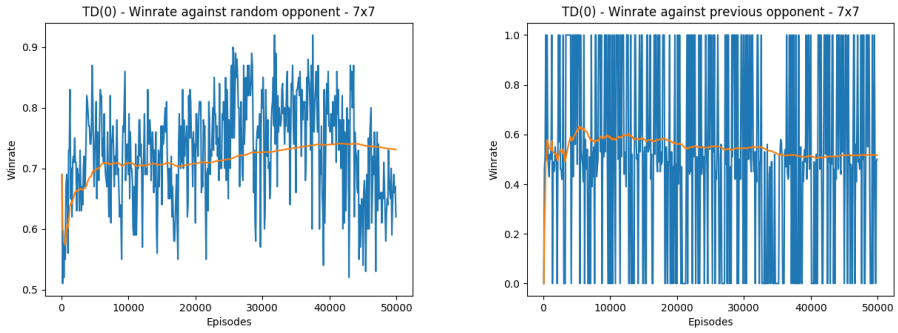


Figure 11: Results from 50.000 episodes of the TD(0) algorithm on a 7x7 board.

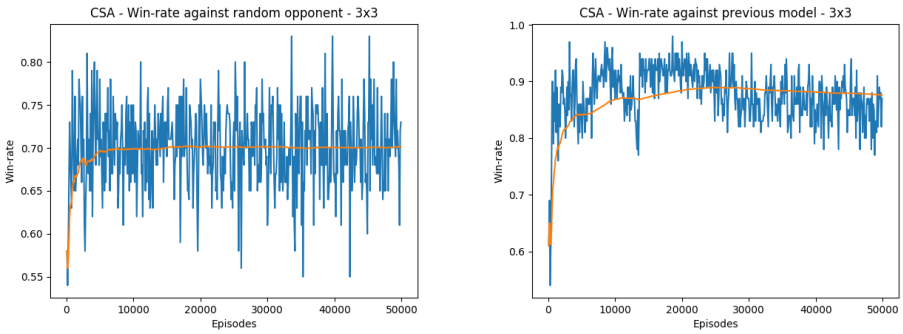


Figure 12: Results from 50.000 episodes of the CSA algorithm on a 3x3 board.

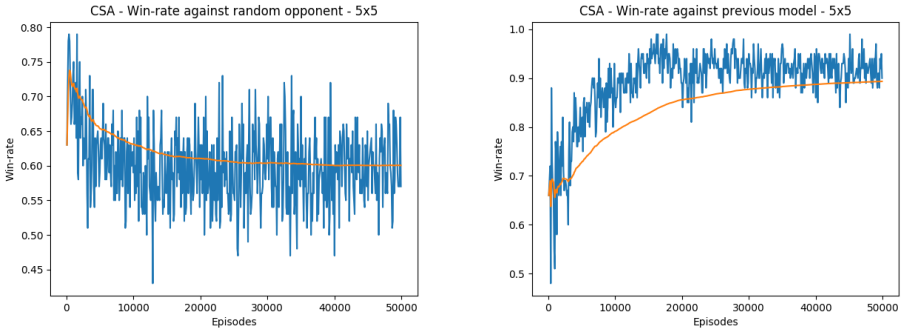


Figure 13: Results from 50.000 episodes of the CSA algorithm on a 5x5 board.

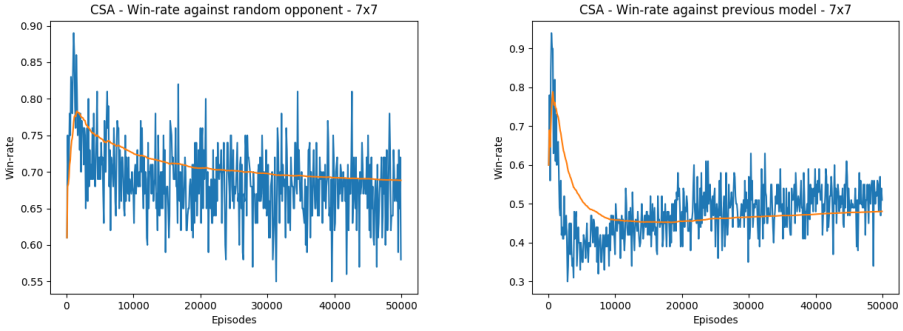


Figure 14: Results from 50.000 episodes of the CSA algorithm on a 7x7 board.

5.2 Activation Function Variation

In this experiment we varied which activation functions the neural-network use. We have chosen to use the `LinearNeuron` which is the default for linear models, `TanhNeuron` and `RectifierNeuron` which are both popular activation functions. For TD(0) we just varied the activation-functions in the input-and the hidden layer because we are using the logistic activation function in the output layer to get values between 0 and 1. In CSA-ES we use the same activation functions in each layer. We only play on 5x5 boards, and we have fixed the neural-network with a hidden-layer with the shape $80 - 40$.
Figures 15, 16 and 17 show the plots generated from training sessions with the TD(0) algorithm with the 3 different activation functions, while figures 18, 19 and 20 are from CSA-ES training sessions.

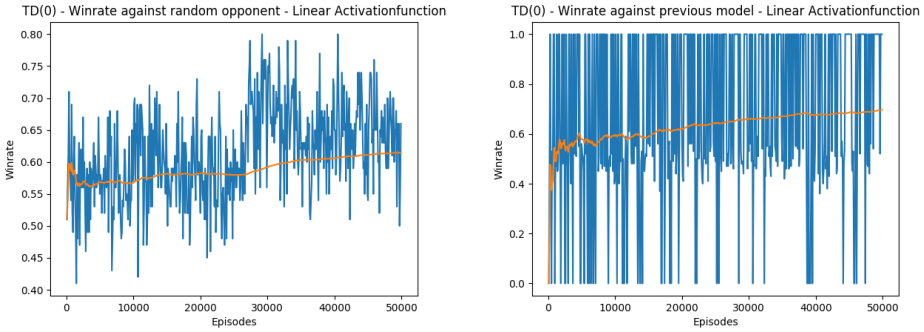


Figure 15: Results from 50.000 episodes of the TD(0) algorithm using the linear activation function.

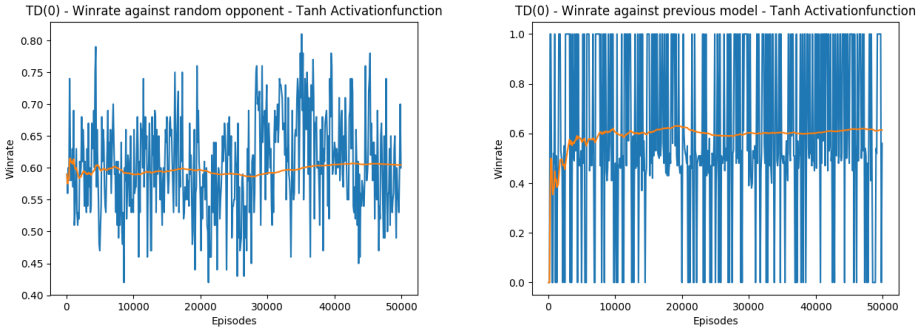


Figure 16: Results from 50.000 episodes of the TD(0) algorithm using the tanh activation function.

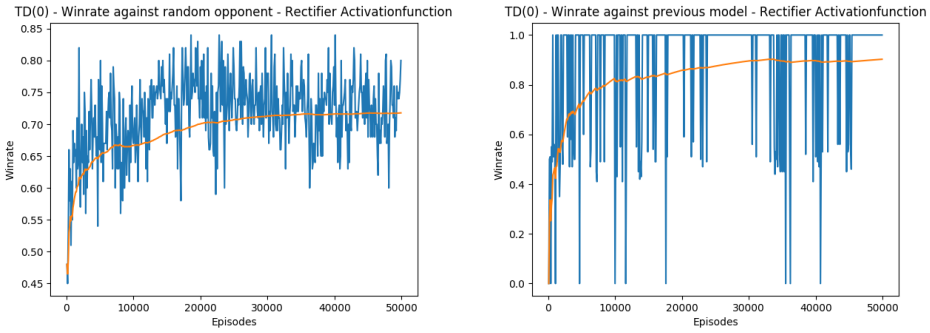


Figure 17: Results from 50.000 episodes of the TD(0) algorithm using the rectifier activation function.

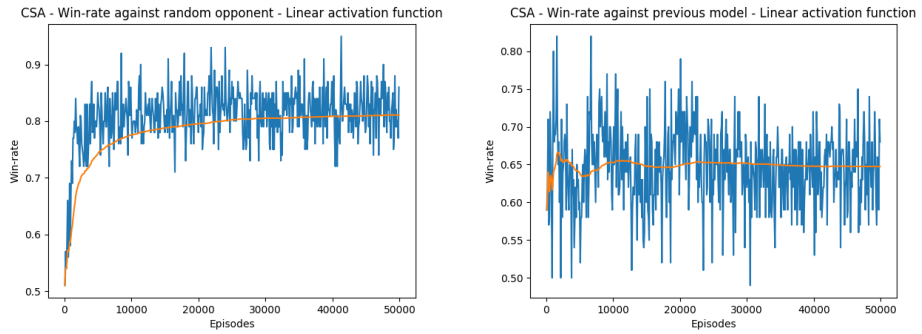


Figure 18: Results from 50.000 episodes of the CSA algorithm using linear activation functions for the neurons.

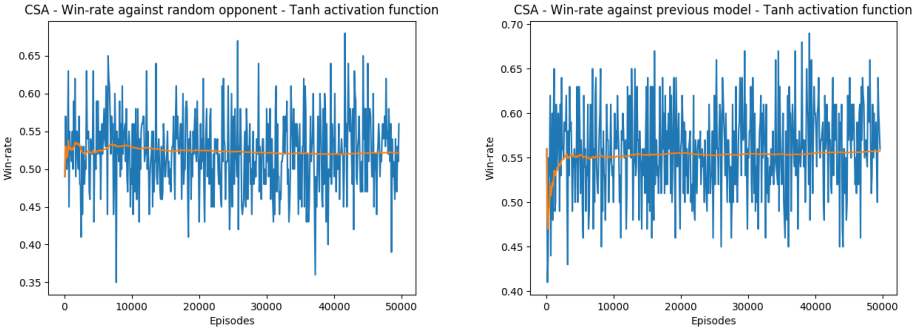


Figure 19: Results from 50.000 episodes of the CSA algorithm using Tanh activation functions for the neurons.

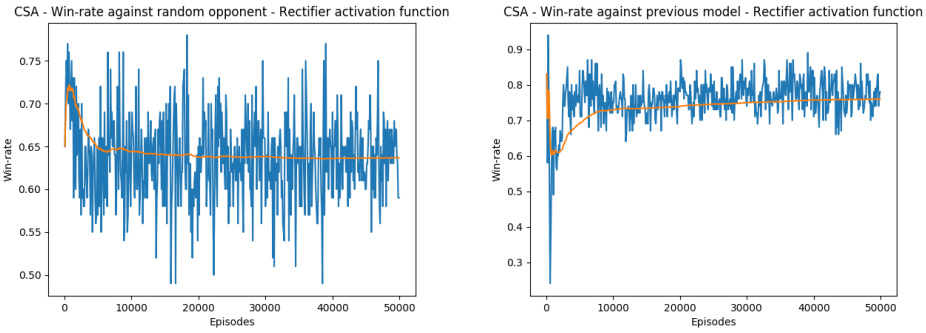


Figure 20: Results from 50.000 episodes of the CSA algorithm using the rectifier activation functions for the neurons.

5.3 Model-Shape Variation

In this class of experiments we have varied the number of neurons in the hidden layer. The neural-network was fixed the `RectifierNeurons` in all layers with CSA and in the first and second layer in TD(0). The experiments were conducted on 5x5 boards. We describe the shape of the hidden as 20-10, which means that the hidden layer has an input of 20 and output of 10, which means that the total number of neurons is $20 + 10 = 30$. We used 3

different settings for the model shapes, 20-10, 80-40 and 160-80.
Figures 21, 22 and 23 show the plots generated from training sessions with the TD(0) algorithm with the 3 different model shapes, while figures 24, 25 and 26 are from CSA-ES training sessions.

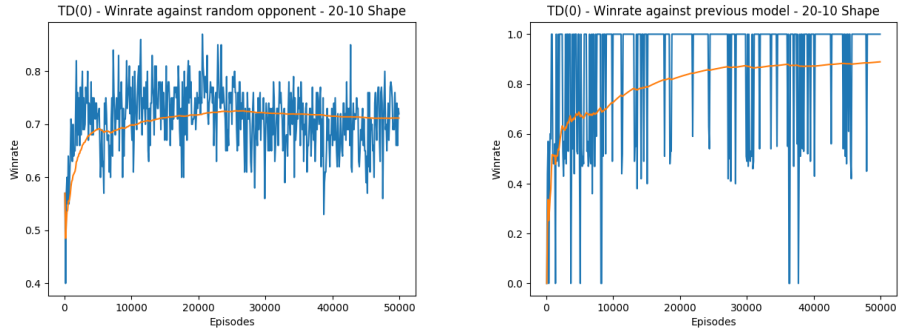


Figure 21: Results from 50.000 episodes of the TD(0) algorithm using a hidden layer with the shape 20-10.

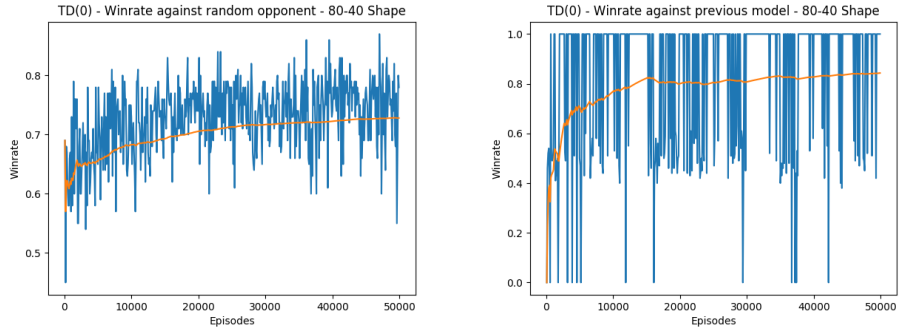


Figure 22: Results from 50.000 episodes of the TD(0) algorithm using a hidden layer with the shape 80-40.

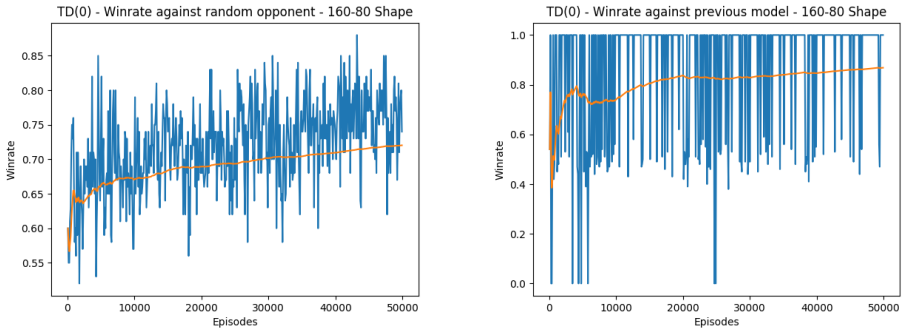


Figure 23: Results from 50.000 episodes of the TD(0) algorithm using a hidden layer with the shape 160-80.

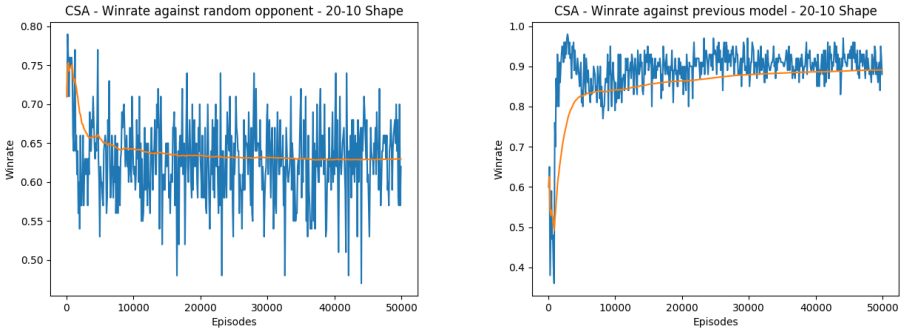


Figure 24: Results from 50.000 episodes of the CSA algorithm, using a hidden layer with the shape 20-10.

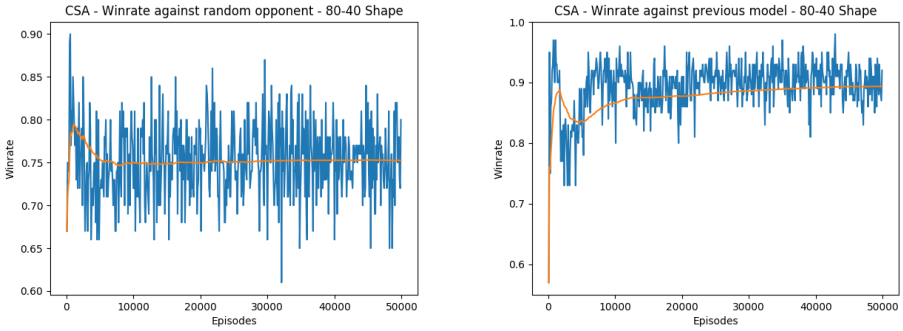


Figure 25: Results from 50.000 episodes of the CSA algorithm, using a hidden layer with the shape 80-40.

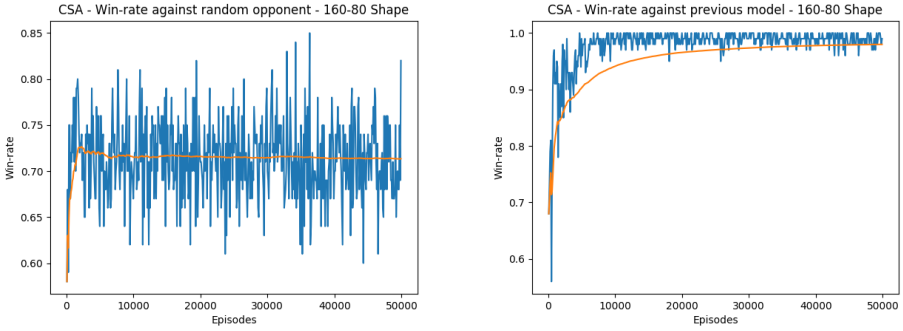


Figure 26: Results from 50.000 episodes of the CSA algorithm, using a hidden layer with the shape 160-80.

5.4 Marathon Session

In this experiment we let the computers run a session overnight with no upper limit on episodes, to see what happens when training many hours. Like in most other experiments the neural-network was using the `RectifierNeurons` in all layers with CSA and in the first and second layer in TD(0). The experiments were conducted on 5x5 boards.

Figure 27 show the plots generated from a marathon training session with the TD(0) algorithm, while figure 28 are from a CSA-ES marathon session.

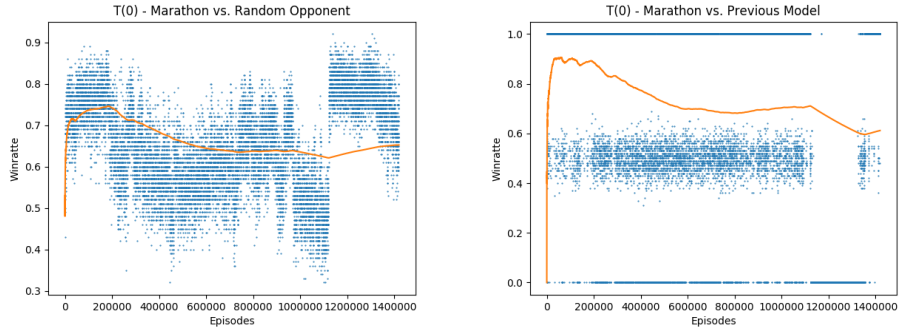


Figure 27: Results from a marathon session with the TD(0) algorithm. Around 1.4 million total steps taken.

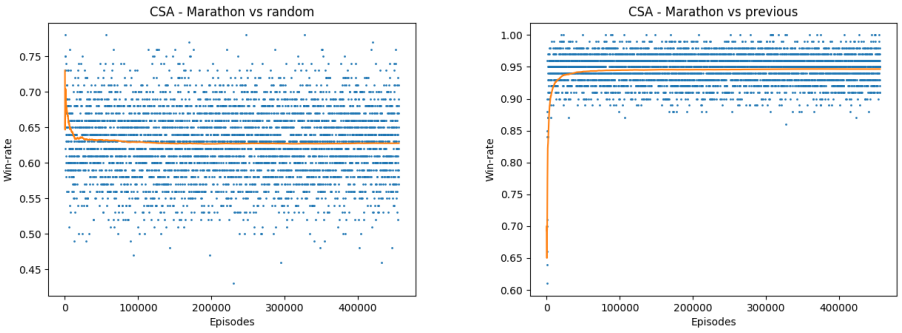


Figure 28: Results from a marathon session with the CSA algorithm. Around 450.000 total steps taken.

5.5 Average of 5 Training Sessions

In this experiment we ran 5 separate training sessions in 5 different processes but with the exact same setup, which is the same setup used in the marathon session described above, except we just let it run 50.000 episodes. We also generated an average of the win-rates of the 5 sessions.
Figure 29 show the plots generated from 5 training sessions of the same setup with the TD(0) algorithm, while figure 30 are from a CSA-ES marathon session.

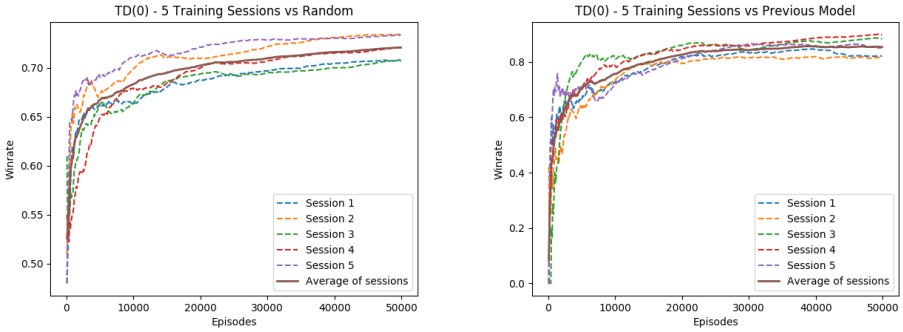


Figure 29: Results from 5 separate sessions with the TD(0) algorithm, each session shown as a dashed graph. The average win-rate is shown as a solid graph.

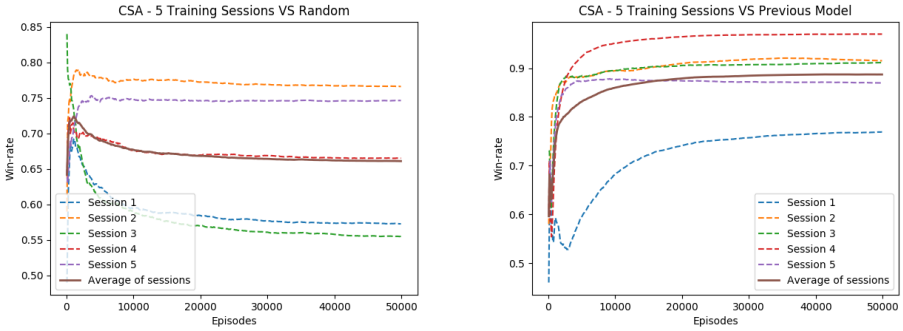


Figure 30: Results from 5 separate sessions with the CSA algorithm, each session shown as a dashed graph. The average win-rate is shown as a solid graph.

6 Discussion

We will start by addressing a significant difference in the results of the win-rate against previous models between the TD(0) and the CSA-ES algorithm. In our TD(0) experiments the win-rate against previous models tend to jump back and forth between 0% and 100%, while this same win-rate for CSA almost never drops below 50% and in some results like the largest model-shape experiment in figure 26 is very stable. Part of this can be explained by a difference in the way our implementation of the two algorithms chooses actions. In TD(0) we only pick epsilon-greedily in training games, that is self-play games that the agent is using to learn from. The purpose of picking epsilon-greedily is to force exploration, so it is not used in the experiment games because we want the best performance of the model. Since there is an equal probability ϵ of choosing the absolute worst valued action as there is to choose the second best valued action it would lead to a lot of bad moves which we don't want when evaluating the model. But because the best valued action is always chosen it means the model is deterministic and therefore very stiff/conservative in it's play-style. In our CSA algorithm implementation however, the action is always chosen by sampling from the set of possible actions according to their values. This means the expected best move is most likely to be chosen, followed by the next best and so on, giving the worst valued move a very low probability of being chosen. This is not as big of a problem as being epsilon-greedy so it is still used in experimental games. In retrospect we could have been more consistent in the implementations, but it is also interesting to see how it affects the results. Because the TD(0) agent is deterministic, when playing 100 games against itself

(or a prior model) it will always win or always loose or it will in some cases depend on who is the starting player, giving the sudden win-rate jumps that is seen in most of our TD(0) experiments. The agent trained with CSA-ES however has a more flexible play-style which is in average better than a totally random play-style at 50%.

6.1 Overall Training Performance

So it seems like we actually training the agents in a meaningful way where they are improving from their random starting points? Looking at our results, most of them seem to indicate that positive progress is being made. We will begin by focusing on the random player experiments. The average win-rate against a random player starts in most experiments around 50% meaning the agents are playing randomly at first according to our baseline test in figure 8. The average is not always around 50% at the beginning, but that is because not enough data is gathered to have a stable average. But after a couple of thousand games the average win-rate starts rising in many experiments. Looking at the experiments which averages 5 sessions in figures 29 and 30 gives a good view of this. Here the average win-rate against a random player is approximately 70% for TD(0) and 67% for CSA-ES. For our TD(0) experiments the deviation in win-rate is very low across the 5 sessions, while there is a significant deviation in the CSA experiments where two sessions ended up with rates as low as 55%, two sessions ended on high rates (one being the best of all sessions of this experiment across both TD(0) and CSA at around 77%) and one session had a win-rate just in between very close to the average of the 5 sessions. This could indicate that CSA is more effected by the starting point than TD(0) is or it could be explained by the more deterministic behaviour of the TD training algorithm where apart from the starting point the only random behaviour is when a move is selected epsilon-greedily. In CSA there's more randomness involved, both in the just discussed action sampling, but also in the way a generation is created by sampling from a Gaussian distribution.
But even though CSA is more unstable across sessions, the average win-rate of both TD(0) and CSA seem to consistently outperform a random player in our experiments, and this is an indication that the agents are in fact in an average training session learning to play the game to a certain degree and are improving from their random starting points. This seems to be the trend across all our experiments. Using the python program `playhex.py` for playing against a saved model we can evaluate the play-style of the models more qualitative. The trend is that the models have found some strategy that leads to wins often with a direct path between each side, but rarely seems to actually respond to the moves that we are making. So even though some learning has occurred, the agent is still at a level of play below the average human beginner-level.

6.2 Convergence

We will now discuss whether the results indicates if the algorithms we have implemented converges. We would expect the algorithm to have converged when the difference in performance between previous agents is small. We measure the performance partly by looking at the win-rate against the previous models. If the win-rate against the previous model is repeatedly 50% it's an indication that the algorithm has converged. Most of our experiments is done with 50000 episodes, which is a relatively small amount if we consider that the state-space, even of a $5x5$ board, is very large. There's not a lot time for the agent to learn how to be good, because it probably won't interact with much of the state-space. The experiment with TD(0), on $3x3$-boardsm, showed in Figure 9 does however appear to indicate convergence, if we look at the total win-rate against the previous model. Here the win-rate against the previous model rises quickly and reaches a maximum at around episode 17000. After that, the win-rate slowly decreases to end up around 50% after 50000 episodes. However looking at the current win-rate in the same plot, it's not so stable. This could be partly because of the deterministic behaviour of our implementation of TD(0) as

discussed. The TD(0)-marathon experiment showed in Figure 27 does look somewhat similar to that of in 9, but the maximum is first reached at around 200000 episodes. This could be due to the larger state-space that in turn requires more episodes. If we stay with $3x3$ boards, with a "small" state-space, the experiment done with the CSA algorithm shown in Figure 12, the total win-rate against the previous model reaches a maximum around 26000 episodes. After that, the win-rate drops slightly for the remaining episodes until ending at around 87%. This indicates that the algorithm has not yet converged, and is actually still improving over the previous models, however after 26000 the rate of improvement appears to dropping steadily. Looking at all other CSA-experiments at bigger board-sizes, it does not appear to indicate convergence. CSA's convergence depends highly on the learning rate α, which is changing between each episode. Over time α gets closer to 0 and as a consequence, CSA improves the agent less and less. However because of the nature of the CSA-algorithm, where only best individuals are selected, it's guaranteed that the next model is generally performing better than the previous. And so what we believe we are seeing, is that the next individuals learn less-and-less "aggressively" (because of the low learning rate). As we are discussing the results, we realize that including the learning rate α and perhaps also the standard deviation σ in the CSA plots would have illustrated this. For TD(0) the learning rate is constant, and what we would expect to see is ere is that the agent should increase it's performance up until a point where it stops increasing, and then oscillate around a maximum.
It should be said that these assumptions of convergence is of course only correct if the algorithms is implemented correctly, because this is when they are guaranteed to converge.

6.3 Experiments With Varying Parameters

Now we will go into more detail with what the results from the experiments where we varied on some parameter can tell us.
In one experiment we varied the board-size while keeping the other parameters the same. The smaller the board size, the easier it is for a random player to "accidentally" win a game. Most actions taken on a small board like 3x3 will place tiles next to existing tiles of the same color, simply because there are so few total tiles, and placing tiles next to each other is the recipe for winning a game of Hex. But smaller boards also mean fewer possible game states which could make learning easier and it is, for a human at least, easy to imagine an optimal strategy for the starting player. An example of this is starting in the middle and then place on the neighbouring tile in the same side that your opponent places his on. This strategy is very simple and guarantees a win after 3 moves.
Looking at the board-size experiment results from TD(0), for the 3x3 board in figure 9 the win-rate against a random player is close to 50% so it looks like it was affected by the higher win chance for a random strategy and does not find an optimal strategy like the one we describe. The win-rates against a random player on the larger boards, 5x5 and 7x7 in figures 10 and 11 were both much better than on the 3x3 board at around 70% and still rising when the experiments ended. Looking at the win-rates against previous models, it seems like it is easier for the agent to improve its strategy on a smaller board like 5x5 where this win-rate is averaging around 80% than it is on 7x7 where the win-rate is around 50% against previous models. Seeing that the win-rate against random is rising a bit however indicates that it is improving a bit, but it is not by much.
The results from the same experiment while training an agent with CSA-ES seems to show a quite different picture. The win-rates for 3x3 and 7x7 in figures 12 and 14 are both around 70% against the random strategy but the 5x5 board in figure 13 only had a final win-rate around 60%. The 5x5 board is however the same setup used in both the marathon game and the average of 5 sessions, and here we saw that it can perform better than 60%, so this session can be considered in the lower range compared to the average. But given this experiment it does not seem like CSA-ES is affected in the same way as TD(0) by the increased

randomness of smaller boards. The win-rates against previous models are comparable to those from TD(0), and it looks like it improves slower on larger board sizes, and again on 7x7 the win-rate is actually below 50%, but it was still rising when the experiment ended. These board size experiments tells us in general that training on a 3x3 board might not be good, which might be explained by it being too easy for a random strategy to win, meaning it will be hard for the agent to find something meaningful and not just keep making random moves that tends to lead to wins anyways. On the other hand training on a 7x7 board seems to be much more inefficient, and these experiments would indicate that it finds a decent strategy that can win a lot of games against a random strategy, but then does not improve much from here. This is probably because the amount of game states rises dramatically for each size increase. Training on a 5x5 board seems to not be affected much by randomness as the 3x3 board, and the rate of learning seems to be much higher than on 7x7, so we recommend this board size for evaluating our training algorithms.

In the second class of experiments the board-size was fixed, and instead we varied on the activation functions- and number of neurons in the neural-network. The main motivation behind experimenting with the activation functions of the neural-networks is to try and determine what works best and perhaps what doesn't work at all. The win-rate for the experiments with TD(0) using the `RectifierNeuron`-activation function is significantly higher than that of `TanhNeruon` and a simple `LinearNeuron` shown in Figure 15, 16 and 17. This is however not the case for the CSA experiments as seen in Figure 18, 19 and 20 where the neural-network with the `LinearNeuron`-activation function, which is actually performing better. Another reason for using activation-functions is to make the neural-network non-linear. The problem that the state-value function, in TD(0) and the action-value function in CSA has to "solve", we think, cannot be approximated with as a linear function. But as seen in Figure 18 in the CSA experiment, the win-rate is actually the best amongst the two other activation function experiments, so we might be wrong about that assumption. We have also tried to vary the number of neurons in the neural-network. The number of neurons in the neural-network(in the hidden layer), together with the activation function, determines how complex the function is that is how complex patterns it can model. So what we would expect to see by using a neural-network with a low number of neurons, is that it will probably require less episodes to adjust the weights in the neural-network, because the complexity of the "behaviour" is limited. If we compare the experiment with the lowest number of neurons seen in Figure 21 with the two other in Figure 22 and 23, it appears that in Figure 21 the win-rate against a random opponent reaches a max at 26000 episodes, whereas in Figure 22 and 23, the maximum hasn't yet been reached, which could mean that there is more to learn. When changing the number of neurons in the neural-network in CSA we can compare with the low number of neurons shown in Figure 24 and higher number of neurons shown in Figure 25 and see that the win-rate is a little higher. However in Figure 26, the win-rate against a random opponent is lower. This could just be due to the fact that there is an overall uncertainty about where the win-rate is going to end up.

6.4 Limitations

The focus of our study is limited in a number of ways. We are from the beginning of the project limited in the general knowledge of the field. This means that much of our work has been focused around getting gathering knowledge about it. This we have done in part by help from our supervisor, and by reading articles and books. The work has as a result of this been, that we have developed our knowledge in an incremental way. In the beginning of the project there was confusion about many things, and as a result things progressed at slower rate. This is also the reason why we ended up implementing TD(0) instead of TD(λ), that was initially our plan. We can say that we started out with higher expectations and less limitations, but we felt us forced to limit ourselves more during the process. We also

initially expected to implement the CMA-ES-algorithm ourselves but we ended up limiting us to using our supervisors implementation of the CSA-ES-algorithm. These are some of the obvious limitations that we have, but there are many things that are left unsaid. One could raise many questions about a lot of the things we have covered in this study, but we have tried, in respect of our limitations, to answers the most important.

6.5 Result Perspectivation

As mentioned in section 2.1 there are very successful examples of reinforcement learning through self-play being used to train agents to achieve master-level play in board games, most notably with AlphaGo by DeepMind [2]. The results of our project are not so extraordinary compared to their success, and the level of play our models have exhibited are more comparable with a 5-year-old than that of a Hex world champion. But this does not mean reinforcement learning through self-play cannot be used to learn to play Hex, in fact we would argue that since our results seem to show a trend of learning to outperform a random strategy and we have used relatively simple reinforcement learning methods, it is an indication that with further research and work a higher level of play could be achieved in Hex. But we don't need to let our project stand on its own, because we have actually found two articles where machine learning was used to train a Hex AI, and one of the papers also used reinforcement learning. In the first paper by K. Young et al. [10] they used deep Q-learning to train their AI, which they call NeuroHex, on a 13x13 board. In the paper they trained NeuroHex for 2 weeks and achieved win-rates of 20.4% as the starting player and 2.1% as the second player against a known conventional Hex AI called MoHex which was the current ICGA Olympiad Hex champion at the time. The other paper by T. Anthony et al. [11] used a novel reinforcement learning algorithm called Expert Iteration (ExIt) where tree search is used to "plan" new policies and a neural network is used in reinforcement learning to generalize those plans. That is in contrast to standard reinforcement learning where a neural network is used to both generalize plans but also to discover them too. Their resulting AI was also able to win against MoHex, so it could seem like they were on to something.
Looking at these examples, it seems like reinforcement learning combined with a form of supervised learning like deep search can give good results, and it could in the future be interesting to see how and if the progress we have made in this project could be used in combination with supervised learning to achieve a higher level of play. However there are also examples of good results from training using pure reinforcement learning, as in TD-Gammon [9] where TD(λ) was used to train an agent to play backgammon. We are unsure however if it would be sufficient as Backgammon and Hex are quite different games. In Backgammon many games are very similar as the pieces are always being moved in the same direction and small changes in the positioning of the pieces does not always have a big influence on the overall game, which is in sharp contrast to Hex where a single piece can change the outcome of a game dramatically. But it could still be worth looking into TD(λ) and see if it would improve our results, which we expect it would to some degree at least.

7 Conclusion

When evaluating the success of our project we can ask several questions. Was our implementations of the simulator and algorithms correct? Were we able to teach an agent to play Hex using reinforcement learning through self-play, and if so did it achieve human beginner-level or even master-level play? Did we obtain the knowledge about the field of machine learning that we were seeking? The final question is hard to answer objectively, but the first couple of question can be answered to a certain degree when looking at the results.

The trained agent consistently, across all experiment, out-performed the random vs. random baseline experiment. This is of course versus a random player, but as we play on relatively small boards, there is a fair-chance of the random playing almost perfect play, and so the agent in general has to make good moves. Because the agent plays better than a random player, does that mean that it has learned to play Hex? Well, we can say for certain that it has learned to play the game of Hex, better than a completely random player. So if we assume that a random playing agent hasn't learned to play the game, an agent who is not playing as random can be said to have learned to play the game more. If we consider the performance against human players(us), we can say that our agents does not learn to play the game of Hex very well, but based on the results of our experiments, we think that it's save to conclude that we have been successful in training an agent to play the game of Hex, at least to a certain degree.
After continuous usage and testing of our simulator during the project period and after conducting a number of experiments, we feel that we can conclude that we have successfully implemented the simulator for the game Hex. The simulator is designed, mostly, in the way we expected, so that it in the future can be used with other algorithms, preferably implemented with the Shark ML-library. We have in this study used the simulator together with two algorithms: CSA and TD(0).
As mentioned in the introduction, part of our purpose with this project was to get acquainted in the field of machine learning with an emphasis on reinforcement learning methods. At the beginning of the project we both had superficial knowledge about the field, mostly limited to knowledge of popular methods used for supervised learning like classification and clustering. We had heard about neural networks and reinforcement learning, but mainly through popular science culture aimed at a wide audience where nothing is explained in great detail. So did we gain the knowledge that we were hoping for? As mentioned it is hard to answer this question thoroughly, but looking at our results objectively we can conclude that we at least reached a point of knowledge where we understood the basics of reinforcement learning and the specific methods we use, TD and ES, to a degree where we could implement something that works. Subjectively we feel we have learned a great deal about the field, and even though we both have had a lot of "Why is this happening to us?" moments, we have had as many if not more "Aha, now I understand!" moments which always gives a great deal of pleasure. But we also feel there is still a lot to learn and there are some details that we have discussed that still lie in a more or less gray area of understanding for us, because we have had a lingering deadline forcing us to move on quicker than what we could have wanted in some cases. In conclusion, our bachelor project has definitely given us a deeper understanding of the field, and courage to explore it even further.

References

[1] D. Gale, "The game of hex and the brouwer fixed-point theorem.," *The American Mathematical Monthly*, vol. vol. 86, no. 10, pp. 818–827, 1979.

[2] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, and T. G. . D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, 2016.

[3] O. Vinyals, T. Ewalds, S. Bartunov, P. G. A. S. Vezhnevets, A. M. M. Yeo, H. Küttler, J. Agapiou, J. Schrittwieser, S. Gaffney, S. Petersen, K. Simonyan, T. Schaul, H. van Hasselt, D. Silver, T. Lillicrap, K. Calderone, P. Keet, A. Brunasso, D. Lawrence, A. Ekermo, J. Repp, and R. Tsing, "Starcraft

2: A new challenge for reinforcement learning." https://deepmind.com/blog/deepmind-and-blizzard-open-starcraft-ii-ai-research-environment/.

[4] R. S.Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, 2018.

[5] O. Krause, "Large-scale noise-resilient evolution-strategies," *In Genetic and Evolutionary Computation Conference (GECCO '19), July 13–17, 2019, Prague, Czech Republic.*, pp. ACM, New York, NY, USA, 9 pages. https: doi.org/10.1145/3321707.3321724, 2019.

[6] Y. Ollivier, L. Arnold, A. Auger, and N. Hansen., "Information-geometric optimization algorithms: A unifying picture via invariance principles," *Journal of Machine Learning Research 18, 18 (2017)*, pp. 1–65, 2017.

[7] N. Hansen, "The cma evolution strategy: A tutorial," *arXiv:1604.00772*, 2016.

[8] C. Igel, V. Heidrich-Meisner, and T. Glasmachers, "Shark," *Journal of Machine Learning Research*, vol. 9, pp. 993–996, 2008.

[9] G. Tesauro, "Temporal difference learning and td-gammon," *Communications of the ACM*, vol. Vol. 38, No. 3., March 1995.

[10] K. Young, G. Vasan, and R. Hayward, "Neurohex: A deep q-learning hex agent," *arXiv:1604.07097v2*, 2016.

[11] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," *arXiv:1705.08439v4*, 2017.

Appendices

A Source Code

The source code for our implementation is attached to this thesis and contained in the folder `hex`. The source code is split into the files: `Hex.hpp` which defines the simulator, `hex_strategies.hpp` which defines the different strategies, `hex_algorithms.hpp` which defines the two algorithms and `main.cpp` which encapsulates the other parts. The source code can be compiled as following, starting in the `hex` folder

```
mkdir build
cd build
cmake ..
make
```

The `hex` executable, and other files related to the compiled project will be built and stored in the build folder. See section 4.4 for a detailed explanation of how to run the program, and what it can be used for.