B.Sc. thesis
Bachelor of Science in Software Development

Generality of Deep Reinforcement Learning for Playing Abstract Strategy Games

Playing Hex with Neural Networks and Tree Search on Consumer Grade Hardware

Magnus Malthe Jacobsen (mjac@itu.dk)

Copenhagen, 2019

IT UNIVERSITY OF COPENHAGEN

Abstract

DeepMind have shown with their AlphaZero algorithm, that it is possible to train a general-purpose deep reinforcement learning agent to gain superhuman playing performance in the games of Chess, Shogi, and Go. I examine if a similar approach, adapted to less processing power, can reach similar results for the game of Hex. Research in general purpose learning agents is of interest because good results in one domain might be translated to good results in others.

During the project several experiments have been run, which in various ways alter the baseline approach slightly. These include: feeding game information to the algorithm in another way; using full pre-activation residual blocks; changing value training target; and mixed precision.

To gain insight into how the various experiments performed, empirical data generated by the experiments are compared to each other, in terms of training loss rates. The game-playing performance of experiments are validated by playing agents against a baseline MCTS agent. The most skilled amongst them have competed in a round-robin tournament with the winner of the 5th Computer Olympiad in 2000 in Hex, Hexy. The empirical data from this tournament has been used to provide Bayesian Elo rating estimations.

Preface

This Bachelor thesis and the project described herein has been produced at the IT University of Copenhagen, as part of the requirements for acquiring a Bachelor of Science degree in Software Development for Magnus Malthe Jacobsen.

The intended reader is a person with some familiarity with computer science, artificial intelligence, machine learning, and deep learning.

Copenhagen, May 15, 2019

Magnus Malthe Jacobsen (mjac@itu.dk)

Acknowledgements

I would like to thank my advisor Troels Bjerre Lund for guidance and suggestions. I would also like to thank fellow students Alexander Mønnike Hansen, Frank Andersen, Mathias Oliver Valdbjørn Jørgensen, and Mikkel Hooge Sørensen, for great discussions and ideas. The people at DeepMind for novel approaches to game AIs and excellent papers describing procedures and results. Furthermore, I would like to thank Vadim V. Anshelevich for granting permission to use his Hex AI Hexy for validation of my trained models. Software acknowledgements:

- **Python** - a versatile programming language that allows for fast prototyping and for me is just a pleasure to work in.
- **PyTorch** - an excellent deep learning framework, that is easy to set up, easy to understand, yet very powerful, very customizable, and very fast.
- **Nvidia CUDA** - again another deep learning enabling software, coupled to the bread and butter deep learning hardware, the GPU. Not because I have directly programmed with the CUDA toolkit, but it has been doing tremendous work behind the scenes in PyTorch.
- **Numpy** - a Python package with many useful numerical and statistical functions, as well as matrix representation of data.
- **Numba** - it felt like it was sent from above, when I discovered numba, a compiler that translates Python code into very fast machine code.

Contents

1. Introduction

Since the advent of the computer science subfield of artificial intelligence (AI) in the 1940s and 50s, solving problems in the form of abstract games, have had a great deal of interest from researchers. At the 1956 summer workshop at Dartmouth College, where the term *artificial intelligence* was coined, participants discussed ways to let a computer agent play chess and checkers, with some form of what came to be known as the *minimax* and *Alpha-beta pruning* algorithms[17, 20].

The interest in solving games was not merely for the amusement associated with the intellectual exercise. Abstract strategy games, such as chess and checker, presents a confined world of discrete actions, with full observability of the game state, and obvious goals to strive for. Because of this, it has proved to be a well-suited environment to apply and test AI systems, systems that might even be applicable to other more complex settings, whether purely abstract or related to the physical.

Early on, researchers split into two camps with different approaches to AI: a more general purpose approach that imitates the human nervous system, what we today call an artificial neural network; and an engineering approach that has a focus on particular tasks and goals for the artificial agent, and apply domain-specifics rules into the systems[17].

General interest in neural networks and self-learning machines from researchers faded in the 1960s and 70s, as the engineering approach showed better results. In the mid-1980s, the proliferation of ever more powerful computers gave rise to a new and increasing enthusiasm from researchers and developers. Although many of the ideas and the mathematical methods go back to the time around the inception of AI, the field is flourishing today, both in terms of research and application. Developments in Graphical Processing Units (GPU) and Tensor Processing Units (TPU) have allowed for faster - and more parallel arithmetic operations, and the development in storage units has allowed for larger quantities of data to be stored[20].

The areas where AI is applied and researched today are manifold, from planning and scheduling, robots, self-driving vehicles, language processing and translation, image analysis, to name a few. However, abstract games are still an area of active research which provides insights that goes beyond the particular game.

1.1. Problem definition

Until recently, the most skilled artificial intelligence agents for adversarial games have depended on built-in domain-specific knowledge. With AlphaZero, DeepMind has shown that it is possible for an algorithm, starting from tabula rasa, to learn how to master the games of Go, Shogi and Chess through self-play[23].

The aim of this project is to expand on the research in this field, by adopting a similar approach as AlphaZero to the game of Hex. Hex, like Chess, Go, and Shogi, is a two-player adversarial abstract strategy game, with perfect information. More specifically, examining the efficiency of artificial intelligence agents, comprised of reinforcement learning, using a combination of neural networks and Monte Carlo Tree Search; trained through self-play; and only with the rules of the game as domain-specific input.

Unlike DeepMind with AlphaZero, this project will make use of general-purpose hardware to run the AI, both for training and playing. Different configurations of the AI on different variations of the game will be tested.

Ultimately, the project will answer the following question: Is it possible for an AI agent, based on the described approach and constraints, to gain a skill-level making it competitive against state-of-the-art game-specific AI agents for the game Hex?

2. Background and Analysis

2.1. Learning agent

When we talk about artificial intelligence in games it is often in relation to the concept of an *agent*. In this paper I follow the definition in Stuart Russel and Peter Norvig's book *Artificial Intelligence - A Modern Approach (3rd ed)*. Here an agent is defined as an entity that acts autonomously to stimuli, adapts to changing circumstances, and strives to achieve a goal in a somewhat rational or optimal fashion[20, pp. 4–5, 15].

The goal of this project is to examine how well an agent can learn playing the game of Hex with an AlphaZero approach, on consumer grade hardware. A very succinct definition of machine learning is from the American computer scientist Tom Mitchell's 1997 book *Machine Learning*:

> *A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E[25, p. 2].*

Using the definition above to outline the problem at hand for the learning agents in this project: The task *T* is to play the game of Hex; The performance *P* can be measured by how well they competes against other Hex AI agents, specifically measured by Elo rating; and the training experience *T* consists of former iterations of the models, playing against themselves.

2.2. Tree Search

For some simple games it is perfectly fine to solve them simply by doing a brute force tree search, from the start of the game, and then examine which action paths guarantees a favourably goal state. An example of such a game is Tic-Tac-Toe. For more complex games, a traversal of a tree consisting of all possible action paths through the state space is impractical. For instance, the American mathematician Claude Shannon gave a conservative estimate of the game tree complexity of Chess at 10^{120}, which today is known as the *Shannon number*.

When doing tree traversal of complex games, some sort of heuristic[1] is often applied, to limit the breadth and depth of the search, and use a value estimation of game states that are not terminal. One problem with this is the *horizon effect* where game states further down the tree, just beyond the "horizon", might lead to a wildly different outcome than the one being evaluated.

Chess has long been the pinnacle of AI research in games. Chess presents a rather difficult task with its game complexity, and enjoys a large degree of popularity around the globe. The first computer chess program to win against a reigning chess world champion, was IBM's Deep Blue, which did so in 1997. An impressive achievement that used all the, at the time, tricks of the trade[2]. But relative to this thesis' problem, it depended heavily on specialized hardware. Its evaluation functions were also very game specific, based on human chess-playing strategies, which were directly implemented into the hardware[4].

For games with a higher *branching factor*[3] than chess, the tree depth that can be reached in a reasonable time is shallower, compared to Deep Blue, using the same techniques. A game like Go is one of those. Go is commonly played on a 19x19 board, which gives a branching factor of 250 for games of average length, compared to 35 for Chess. For this reason researchers came up with a traversal technique that grows a game tree asymmetrically, but still finds a balance between exploration and exploitation, breadth and depth. This has come to be known as *Monte Carlo tree search* (MCTS). In the original version, evaluation of game states is being done by a approximating a value through random roll-outs to a terminal state. Because of this, MCTS can be said to be of a more general nature, compared to tree search techniques that use domain-specific evaluations. When the number of times a single game tree is traversed goes towards infinity, it is proven, under some conditions, that the algorithm will converge on an optimal action[8]. MCTS was successfully applied to the game of Go, but still without acquiring superhuman skill level.

2.3. DeepMind's AlphaZero

Because of the historical primacy of agents with domain-specific heuristics built into their algorithm, it was an astonishing feat when the Alphabet Inc. owned company DeepMind published results of their general purpose reinforcement learning algorithm AlphaZero. This algorithm was able to achieve superhuman game playing skill, starting from tabula rasa, and only training on data generated by self-play[23].

AlphaZero is a continuation of DeepMind's Go playing algorithm AlphaGo which gave Deepmind popular fame when it was the first AI agent able to beat a professional Go

[1] A function that guides search.

[2] It implemented quiescence search, iterative deepening, transposition tables, NegaScout, an opening book, endgame book, among other things.

[3] The average number of legal moves in a game of average length.

player of the highest rank, South Korean Lee Sedol, in 2016[24]. AlphaZero in contrast to AlphaGo is more generic, and results showed that it was able to master several different abstract strategy games: Chess, Shogi, and Go[23].

The AlphaZero algorithm uses a single deep Residual Network (ResNet) model, which is initialized with random weights and implemented by the use of the machine learning library Tensorflow. During execution, when the agent is tasked with suggesting a move, it uses a MCTS, to explore a game tree. Here the model is fed game state information translated into an image[4], to both predict the value of a game state to guide exploitation, and a policy of actions from the game state to guide exploration.

During training of the models, for each of the games Chess, Shogi, and Go, Deepmind used 5,000 1st generation *TPU*s to generate game data. TPU stands for Tensor Processing Unit, and is Google-engineered hardware specialized for neural network machine learning in their server farms, in particular fast matrix multiplication with low energy consumption. It is difficult to find performance information for this specific piece of hardware, but the 2nd generation TPU has a theoretical computational performance of 45 TFLOPS[5][15]. For the game of Chess they spent 9 hours of training and game playing, 12 hours for Shogi, and 13 days for Go on a 19x19 board[23]. Had they run it on a single TPU, instead of the 5,000, that would have amounted to ≈ 5, ≈ 7, and ≈ 178 years respectively.

2.4. Consumer grade hardware

In many parts of machine learning, and deep learning in particular, normal Central Processing Units (CPU) do not suffice. What is needed is hardware that is capable of loading large quantities of data fast, getting access to said data quickly, and perform many operations in parallel. For this reason a Graphical Processing Unit (GPU) is by far superior to a CPU. However, there is still some significant latency overhead associated with transferring data to and from a GPU, which one will have to consider when building applications that utilize the GPU heavily.

In reference to the problem definition, a challenge in this thesis project is to only use consumer grade hardware, therefore specialized hardware like a TPU is out of the question. The two main GPU manufacturers Nvidia and AMD both separate their microarchitecture product lines in consumer and professional, where the former are sold in retail. For parallel computing Nvidia has been quicker to support programming these sort of tasks on their GPUs, through their CUDA platform. As a result Nvidia GPUs are much more supported in machine learning libraries, than AMD GPUs.

For Nvidia's most recent microarchitecture targeting consumers, the GeForce RTX se-

[4]Here an image should be understood as game state features converted into a multi-dimensional binary array.

[5]FLOPS: Floating Point Operations per second. TFLOPS = 10^{12} FLOPS

ries of GPUs all include cores specialized for multiplication of lesser precision matrices (half-precision float, int8, or int4), so-called Tensor cores[19]. For the reasons outlined above, for running the neural network models in this project, I will be using a Geforce RTX GPU. Specifically the 2060 version, for practical reasons, as that is the RTX GPU I have access to.

Unlike DeepMind, I do not have access to a multitude of individual processing units, which I will have to take into consideration when designing the algorithm. However, as an individual piece of hardware and with the challenge in mind, the RTX 2060 should perform reasonably.

2.5. Hex

The game of Hex was invented by the Danish mathematician and poet Piet Hein in the early 1940s. A few year later it was independently re-invented by the American mathematician John Nash[13].

Hex is a game of charmingly simple rules, but with complex strategies. It is a connection game, on a board of equal length sides, where the positions have hexagonal edges to other positions unless they border the boundary of the board. Each player has two opposing sides as "their" own. At the start of the game, the board is empty and the players alternately place their individual markers on unoccupied positions on the board. The goal is to make a connection between one's two sides. Often the game board is displayed as either a flat or diamond shaped rhombus. An example of the latter can be seen in figure 2.1, where the light blue agent has won, by constructing a connection between its two sides.

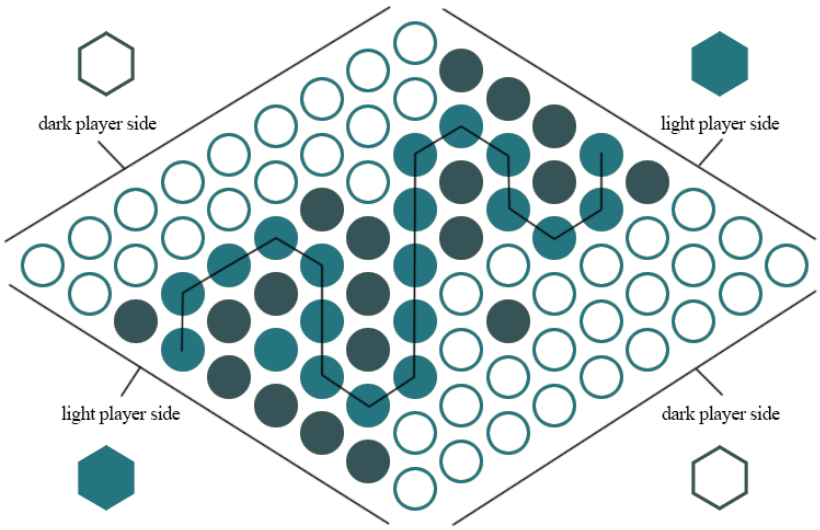

Figure 2.1: A 9x9 game where the light blue agent has won, as its two sides, the left to bottom and the top to right, are connected. The connection is illustrated with a line, showing the unbroken connection between the two sides.

With the above-described rules, the first player has a strong advantage. As such, it is not

uncommon to employ a *swap rule*, where the second player, after the first player's initial action, has the option to swap and become the first player and transpose the position of the piece. This evens out the first player advantage. However, since I intend to keep the game part of the project as simple as possible, I will ignore the swap rule.

2.5.1. Complexity

Because of the similarity of the rules of Hex and Go, in terms of how the board is configured and their turn based nature, it also exhibits a similar game complexity, when equal board sizes are used. Complexity specifically in the form of a rather high branching factor.

An advantage of Hex is that the simple rules allows for a simpler software implementation of the game, and said rules allows for adjustment the complexity by either decreasing or increasing the board size.

Increasing the size of the board in Hex, similar to Go, is not trivial, when looking at how it affects the game tree complexity. If one estimates an average Hex game length to be 40% the number of board positions, that would amount to ≈ 32 for a 9x9 board, and ≈ 20 for a 7x7 board. The average branching factor would then be $81 - 32/2 = 65$ and $49 - 20/2 = 39$ respectively. We can then roughly estimate a lower bound of the game tree complexity, as the average branching factor to the power of the average game length: $65^{32} \approx 1.03 \times 10^{58}$ for 9x9, and $39^{20} \approx 6.63 \times 10^{31}$ for 7x7. A quite significant increase.

2.5.2. Strategies and AI agents

An early Hex playing AI agent was constructed by Claude Shannon and Edward F. Moore, and described by the former in a paper in 1953. To put it in the author's own words, it was among the type of "*Machines applying general principles of approximate validity*"[22, pp. 705–706]. Their agent was interestingly built as an analog device, that functioned as a resistance network, with electrical resistors as edges and light bulbs as positions. One player's pieces had positive charges, the other negative. This device proved to play pretty well against humans, especially when it was the starting player. It did suffer from being unable to come up with reasonably good end-game combinatorial plays[22].

The winner of the 5th Computer Olympiad in 2000, in Hex, was Hexy, an agent created by Vadim V. Anshelevich. It uses an evaluation function similar to Shannon and Moore's device, but enhanced by a theorem proving algorithm that finds virtual connections. Virtual connections, as opposed to realized connections, are board subpatterns, where one player is guaranteed to be able to create a connection, no matter how the opposing player plays in that subarea, given they play optimal[2].

An illustration of 3 simple virtual connections, known as two-bridges, can be seen in figure 2.2. Even though it is the dark player's turn, it can not prevent the light player from connecting its two sides, if the light player plays optimally. Each of the letter-marked position pairs **ab**, **cd**, and **ef** performs a two-bridge, where if the dark player places a piece in one of the positions, the light player can place a piece in the corresponding position of the pair. It should be noted that Hexy also looked at more advanced virtual connections, and that the evaluation function was applied in a alpha-beta search algorithm.

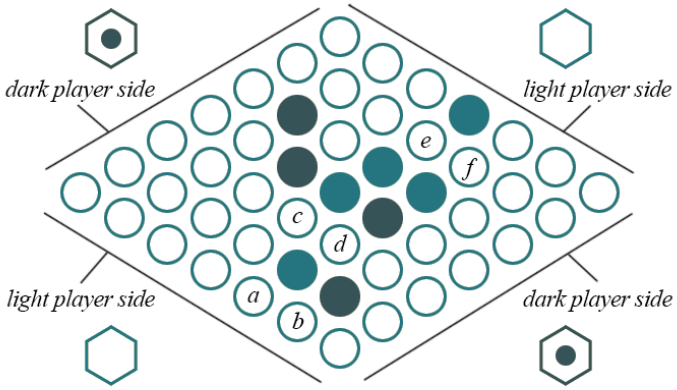

Figure 2.2: Illustration of 3 ***two-bridges****.*

Among the most successful newer Hex-playing agents have been MoHex and MoHex 2. MoHex, among other approaches, combines Hexy's method of evaluating virtual connections with MCTS. MoHex 2 is a further extension of MoHex, where the MCTS search is further aided by a supervised machine learning algorithm that is trained on win correlation of board patterns. In the paper describing MoHex 2, the creators interestingly state that according to their findings, an algorithm with only MCTS and some other features, but missing the explicit virtual connections evaluation, borrowed from Hexy, perform rather badly. They suggest that their MCTS simulation policy is unable to observe global virtual connectivity[13].

A final Hex playing agent I want to mention is EXIT[3]. Like AlphaZero it also uses a reinforcement learning neural network, starting from tabula rasa knowledge. However, it is closer to AlphaGo Zero, in that it employs separate neural networks for value and policies. The EXIT algorithm were able to beat MoHex on some different settings[3].

2.6. Validating performance

To find out if the trained agents have actually learned anything, some game-playing performance measuring is necessary. A very simple way of doing so is to measure win rate against a baseline agent. For this I intend to use a simple non-neural network MCTS implementation, playing with a significant amount of simulations.

Another more complicated way is to measure some sort of Elo rating among the agents, as they play against each other. To be able to answer the question of whether my learned agents are able to acquire a state-of-the-art playing skill, I think including other agents than the baseline MCTS is required. Therefore I intend to use Hexy, on expert settings, against the most promising of the experiments from this project, in addtition to the baseline MCTS. I would also have liked to include newer developed Hex playing agents, like MoHex 2 and EXIT, but were not able to find runnable versions of them.

To estimate ratings, I intend to use a freeware tool by French computer scientist Remi Coulom, that estimates Bayesian Elo ratings[7]. Bayesian Elo rating, which in difference to more regular Elo ratings, takes uncertainty, in regards to sample size, into account, and the specific implementation also takes information about which player is first, into account. Even though the program is originally developed for Chess games, I think it is suitable for the task.

2.7. Recapitulation

To sum up the background and analysis chapter, I will return to my adaptation of Tom Mitchells machine learning definition, and now elaborate on each of the three parts.

The task *T* at hand is still to play the game of chess. But considering that DeepMind spent what is equivalent to 178 years of TPU time while generating 19x19 Go game data, I find it reasonable that I, with my single RTX 2060 available, scale the Hex boards, I will train models on, down. Additionally, for the sake of simplicity of the implementation, I will also ignore the swap rule. How the algorithm executes the task will be inspired by the AlphaZero approach that makes use of both a Residual Neural Network for value-policy inference, and a Monte Carlo tree search that looks for good moves, in accordance with what the network returns.

The performance *P* will be measured against other trained models, as well as a baseline MCTS agent. However, since pure MCTS is not that good at playing Hex, as stated in the Mohex 2 paper[13], I will also use Hexy to match up against.

The experience *E* will be generated through self-play, and starting from tabula rasa. In addition to a scaled down board size, I don't expect to be able to generate the same amount of games that DeepMind has been able to. Again, because I have to take the smaller amount of computational power available to me into consideration, it is interesting to use the Tensor cores that the RTX 2060 provides, which will require network models that use half-precision floats. This should allow me to speed up the process of generating game data, and as such more data to train on should be available.

3. Methods

3.1. Residual Neural Network

The selected form of neural network is a Residual Neural Network (ResNet). A ResNet allows for much deeper networks compared to other popular network types, when it was first proposed[11]. Other networks had a tendency to actually become worse at predicting, that is getting a higher error rate in validation, when they were built deeper. Deeper understood as having more layers added. This is known as the so-called *degradation problem*, where gradients[1] would either vanish to zero, or explode to very large numbers. The proposed solution was to make layers a part of the building block that also contained a shortcut function, also called skip connection, where the input would be identity mapped, and then added to the result of the other part of the function.

An illustration of the originally proposed residual block can be seen in figure 3.1. What is not included is a batch normalization function applied after each weight layer. ReLU stands for Rectified Linear Unit and is equivalent to $ReLU(x) = max(0, x)$. The weight layers are convolutional layers.

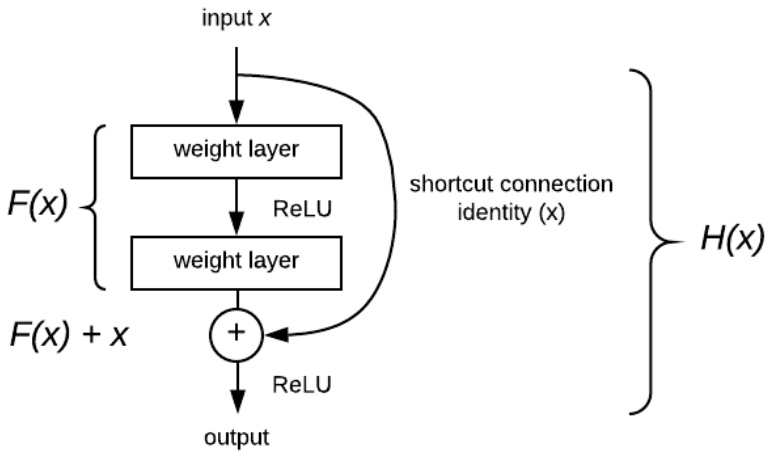

Figure 3.1: The original residual block design.

If we conceptualize what we want from a part of the neural network, it is to get a desired mapping $H(x)$. This can then reformulated into a residual function $F(x) = H(x) - x$.

[1]Gradients are vectors consisting of the partial derivatives of the layer functions. A gradient points in the direction of the greatest increase of the functions' curves.

Since we are still interested in getting the desired mapping, $H(x)$ we can cast the function to be $H(x) = F(x) + x$. This might not look like much, but it showed much better error rates, compared to state-of-the-art network frameworks at the time of publication of the paper, when trained in image classification. Furthermore, it reversed the negative effect when building even deeper networks. With a ResNet, their results suggested that the deeper the network the lower the error rate. The authors hypothesized that it is easier to optimize a residual mapping, $F(x)$ than the entire desired mapping $H(x)$[11].

The same authors did further experiments, among them on the function used as skip connection and the design of the residual block. The best configuration was to keep identity mapping for the shortcut, but do both activation functions, batch normalisation and ReLU, prior to the weight layers. This they called a *full pre-activation*, and can be seen in figure 3.2[12]. Their conclusion on this is that doing batch normalisation prior to the weight layers improves regularization, which helps in reducing overfitting[2].

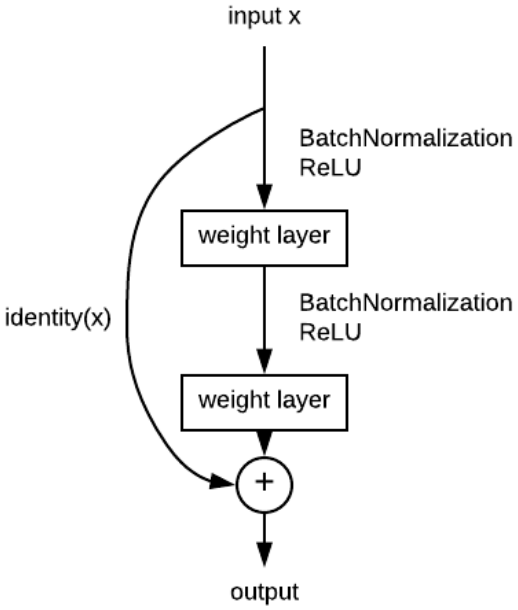

Figure 3.2: Full pre-activation residual block design.

In AlphaZero, DeepMind used the original residual block design. I intend to experiment with both the original and full pre-activation, to see if there are any noticeable differences in results.

3.2. Network anatomy

The anatomy of the neural network used in AlphaZero is defined in the paper "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play"[23]. A basic outline can be seen in figure 3.3.

[2]Overfitting can be seen as a form of memorization. It is when a network model has specialized on predictions on specific training input, but is not learning general patterns.

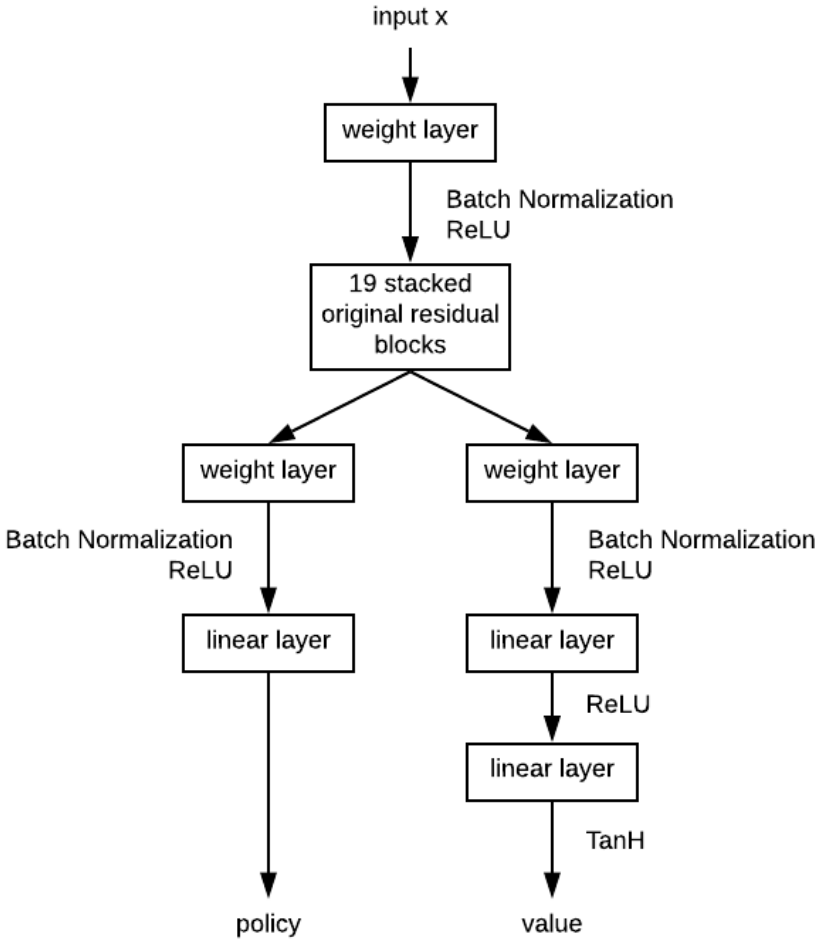

Figure 3.3: AlphaZero's neural network structure.

As said in the previous section, I intend to test the effect of replacing the original residual blocks with full pre-activation ones. Considering that I have less computational power at hand, it would also be interesting to measure the effect on reducing the amount of residual blocks.

3.3. Monte Carlo tree search

Roughly speaking, when Monte Carlo tree search (MCTS) was introduced, the intent was to use random sampling in a tree search, to solve decision making with the goal of getting close to an optimal action. When deciding which child node of a parent node to visit during a tree traversal, it is common to utilize the argmax[3] on the Upper Confidence Bounds (UCB1) formula applied to each child node.

$$v_i + c \times \sqrt{\frac{\ln N}{n_i}} \tag{3.1}$$

In equation 3.1, v_i is the value of the child node, N is the amount of times the parent node has been visited, n is the amount of times the child node has been visited, and c is a

[3]A function that returns the largest element of its input.

constant that controls the importance the square root part of the formula. This formula serves to balance investigating parts of the tree that promises high value outcomes and parts that er under-explored. This is known as the exploitation versus exploration. The balance between the two can be adjusted by changing the value of the constant c.

In 2006 Hungarian computer scientist Levente Kocsis and mathematician Csaba Szepesvári published an MCTS algorithm that used random rollouts[4] for approximating values of nodes, and a minimax strategy for backpropagating values up the tree. The idea is that when enough random rollouts are performed it provides good approximations for the value of the different choices at the root level. This is because UCB helps building the tree asymmetrical, balancing the proportion of exploration and exploitation. This basic algorithm is known as UCB applied to trees (UCT)[16]. In figure 3.4 one can see an illustration of the workings of this algorithm, inspired by a similar created by Guillaume Chaslot et al.[5].

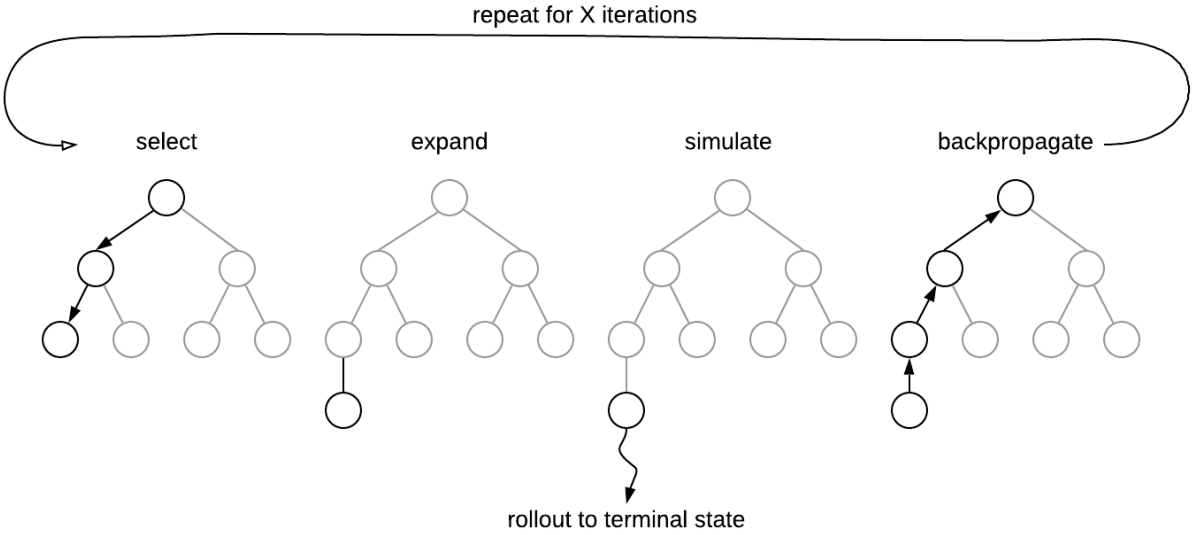

Figure 3.4: Outline of the UCT algorithm.

As shown, the algorithm works in 4 main steps. The first, *select*, runs recursively with argmax and UCB1 used to choose a new parent node, until we reach a node that has at least one child node that is unvisited, or we reach a terminal state node. In *expand* we return the given node if its state is terminal, otherwise we choose one of its unvisited children. In *simulate* we do random rollouts till we reach a terminal state. In *backpropagate*, we go back up the tree, incrementing visit count by one for all the nodes in the path, and in a minimax fashion add the value found by *simulate* to a value accumulator for each node. The minimax value update is usually done by flipping the value to the perspective of the player who took a turn to reach the node. The value used in their UCB1 is the average value of a node, the accumulated value divided by the amount of visits[16].

[4]A stochastic simulation that recursively takes random actions until a terminal state is reached.

3.4. MCTS with neural network

In AlphaZero a variant of Polynomial Upper Confidence Tree (PUCT) is used, which itself is variation on UCT. In addition to some changes to the UCB formula, the expand and simulate steps are combined into a single *evaluate* step. In the evaluate step, the network is given a feature image array for the board state, and then returns a value for the state, and a policy vector based on prior visits to children nodes, i.e. actions taken from that state. The policy vector is then reduced to only contain the legal moves, and then applied a Softmax function. After this, the policy values are saved for the corresponding child nodes as *prior* visit values.

$$C(p) = \log\left(\frac{1 + N_p + 19652}{19652}\right) + 1.25 \tag{3.2}$$

$$U(p, c) = C(p) \times P_c \times \sqrt{\frac{N_p}{1 + N_c}} \tag{3.3}$$

$$UCB(p, c) = \frac{V_c}{N_c} + U(p, c) \tag{3.4}$$

Equation 3.2 corresponds to the c exploration parameter from the original UCT. N denotes visit count to a given node, p denotes a parent node. It starts at $C \approx 1.25$ under the first iteration, and grows ever so slightly. In AlphaZero's implementation the number of MCTS iterations is set to 800, making the equation max out at $C \approx 1.29$[23]. Given how small the increase is, I find it reasonable in my own implementation to set it to a constant value, thus avoiding doing the logarithmic calculations. I suspect that this difference will not hamper the exploitation versus exploration of my algorithm in any major way.

In equation 3.3, p and c denotes parent and child node respectively. P is the prior value from the policy output of the neural network. In equation 3.4, V denotes the accumulated value of a given node, and the entire equation gives the UCB value for the child node. It corresponds pretty well to equation 3.1, with the main difference being the prior value added as a weight to the exploration part of the equation. Thus the neural network is informing the UCB formula used in MCTS, on how much it should weight exploration of particular nodes, and how they should be evaluated.

3.5. The algorithm

Self-play is essential for the algorithm, as it generates data for the network model to learn from. A rough outline of the running of the algorithm, imagined as if it was run sequentially, can be seen in figure 3.5. During self-play the AlphaZero algorithm behaves

like the variant of PUCT described in the previous section, but with some random noise added to the priors of the root's children nodes, and a Softmax sampling for certain steps of each game.

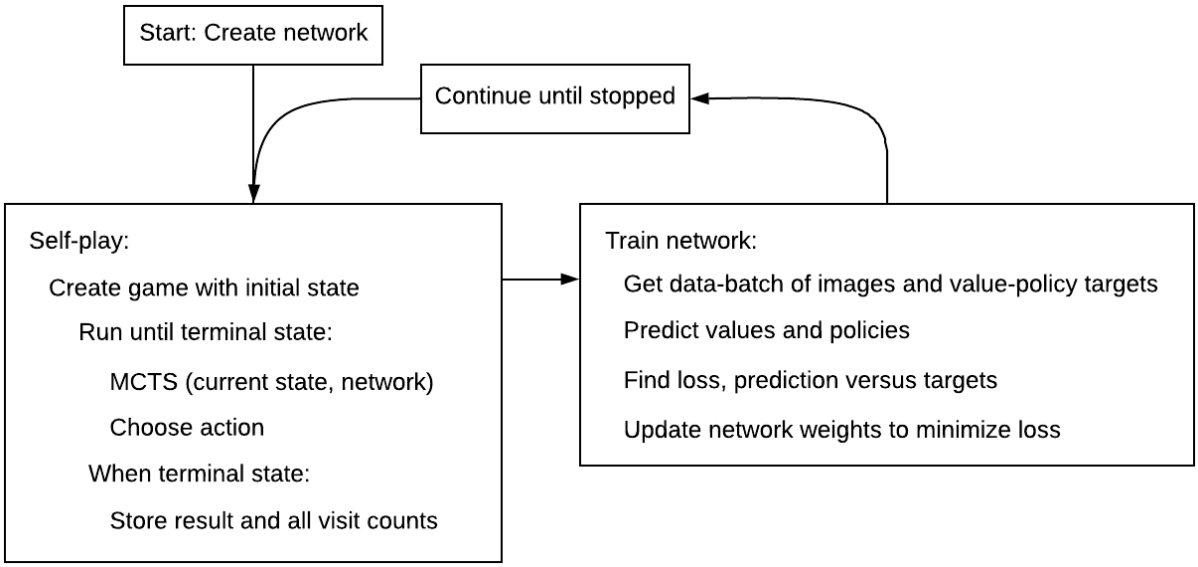

Figure 3.5: Outline of the AlphaZero algorithm.

3.5.1. Self-play

The network model's prediction might be flawed at certain steps in time, which could lead the model to converge on suboptimal action paths, and then keep training on those. Or it might be used to only seeing optimal plays, and thus will not be able to handle situations where its opponent plays suboptimal. To ensure some varieties in the game states seen by the model during training, for a certain amount of initial steps of each game, the actual action taken by the agent is chosen by a probabilistic random selection, using the result of the child node visit counts fed to a Softmax function. This random selection does almost not affect the saved data, as it does not impact the prior visit values. One could argue that it somehow flaws the value data indirectly, as the random selection might obscure who ends up winning the game. This addition to the self-play helps the network model to experience a larger state space, thus making it more robust.

For each action for the algorithm to decide, randomly generated noise is also mixed into the prior values of all the root node's children. A so-called *Dirichlet distribution*, which consists of normalized gamma-distributed probabilistic values. This also diversifies the horizon of what the MCTS sees under different runs, making it possible to observe potentially stronger paths that a current network would otherwise advise against. AlphaZero uses different α-values depending on the game; 0.3, 0.15, 0.03, for Chess, Shogi, and Go respectively. In contrast to the probabilistic random Softmax selection described in the paragraph above, this impacts both value training targets and policy training targets, because the noise affects which nodes are visited, and thus also the backpropagated values[23].

3.5.2. Training

During training, the algorithm randomly samples board states, among recorded self-play games, at randomly selected steps in time for each game. Images are created from these board states. They are then paired up with the corresponding information about the end result of the particular game, and how the visit count distribution looked for states' child nodes, after MCTS had been run on that state. The network is then tasked with making predictions on these images. A *Mean Squared Error* function is applied to the predicted value and the target value. A *Softmax Cross Entropy with Logits*[5] function is applied to the predicted policy and the target policy. The model is then optimized with the combined loss, using Gradient Descent.

3.5.3. Training targets

In regular UCT MCTS, information about the actual game, the value of a state, is fed into the algorithm through the rollouts. In the AlphaZero algorithm the bread and butter for learning the game is training on previously played games. Here network inference is guiding the MCTS, which again provides the network information about visited root child nodes, and together with the game's end result, the utility for the terminal state of a game, forms targets the network optimizes to predict. The most important factor is this utility, as that is when information about the real game is fed into the algorithm. An illustration of this can be seen in figure3.6.

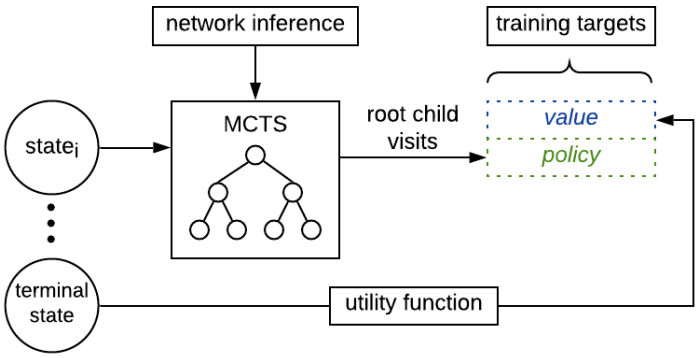

Figure 3.6: Outline of how training targets are generated in the original AlphaZero approach.

Developers from Oracle have experimented with alternative ways of giving an AlphaZero-like algorithm information about state values[1]. They test several approaches, of which I will describe two.

First, they use a game's utility function to give a value when the MCTS part reach a terminal state, thus bypassing the network, in order to get real game information. This strengthens the approximation, especially early in the run of the entire algorithm, when the network is still affected by the random weight initialization. It also indirectly helps the network, since better evaluations gives a more optimal visit count, which gives

[5] I have not been able to find the Tensorflow implementation of this.

better policy for the network to optimize on. An illustration of this approach can be seen in figure 3.7.

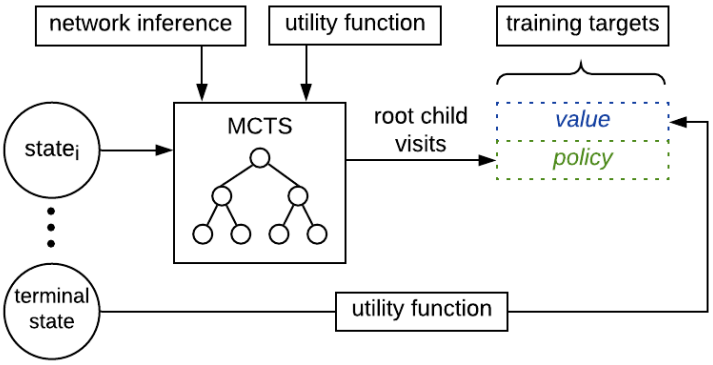

Figure 3.7: Outline of the approach with utility used in MCTS evaluation at terminal states.

Second, they change the training target value for a given board state, from the utility of a games terminal state, to the average value approximated by MCTS for the specific state in the specific game. This does not work if it is independently applied to the original AlphaZero approach, as no information about the real game utilities would then reach the algorithm. A solution like the one described in the previous paragraph is needed. This is illustrated in figure 3.8. The combined approach allows for the target value of a state to be a decimal number in the vicinity of the range -1 to 1. The original approach uses discrete integer value targets, for chess -1, 0, and 1, even when a state is far from a terminal state, and one would assume there is still some uncertainty on the outcome.

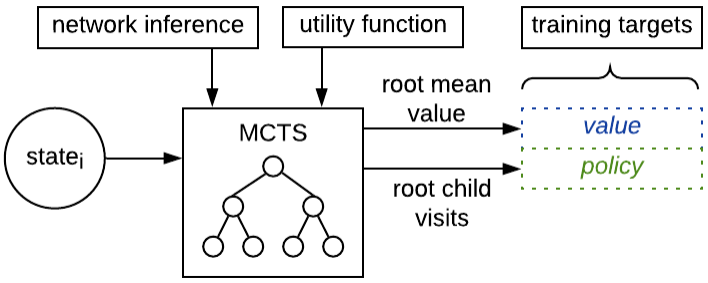

Figure 3.8: Outline of the approach with both utility used in MCTS evaluation at terminal states, and the mean value at root level as value training target instead of the final game result.

3.6. Half-precision

Half-precision floating points are of interest for deep learning, because they allow for twice the amount of variables being stored in memory, compared to single precision. But also because the new line of consumer-minded Nvidia GPUs contain Tensor cores, which allows for faster half-precision matrix operations. The hardware that will be used in this project includes such a GPU.

The most common standard for half-precision floating points is the *binary16* from the IEEE Computer Society, which only uses 16 bits to represent a number[6], compared to 32 bits for a regular single precision .

Half-precision floating points take up half the amount of storage, at the cost of precision, compared to single precision floating points. For half-precision, the largest representable number is ≈ 65504 , and the smallest positive representable number is ≈ 0.000000059605. In the range between these two, there is a lot of imprecision, even for the left side of the decimal separator, where not all integers in the range 0 to 65504 can be represented. Thus, converting from single to half precision, one can loose accuracy, as some sort of rounding occurs.

When one does an arithmetic operation on a half-precision number, that makes it smaller, but still positive, than ≈ 0.000000059605, it will round down to 0. If one tries to reach a number too much greater than ≈ 65504, it will result in ∞. Furthermore, doing $\frac{\infty}{\infty}$, even if one or both has a negative sign, results in *NaN*.

Researchers from Baidu Research and Nvidia have published a paper, where they discuss results from doing mixed-precision training for different common machine learning tasks with popular deep learning model frameworks, ResNet included[18]. Their mixed-precision model has a single-precision master model, on which all optimizations are done. Under each training iteration, a copy of the master, with weights, activations, and gradients being converted to half-precision. In order to avoid vanishing gradients or exploding gradients they perform a scale on them, when it is required. Additionally, all accumulations occur in single precision. Their results showed increased speed, and in some cases an increase in accuracy. The latter they speculate is because the conversion functions as a form of regularizer on the weights.

4. Implementation

A requirement of mine for the software environment was that it allows me to easily debug and implement changes, and that there is a significant support for the chosen language and frameworks. Personal preferences in style have also weighted in.

4.1. Software environment

I have chosen Python as the programming language for the implementation. A compelling argument is that it has more or less become the language of machine learning, especially for programmers; with lots of libraries and packages available, as well as online tutorials. It also allows for very laconic expressions, which suits me well. A compelling counter argument for why this is not an optimal language for deep learning, and especially in this project where computational power is limited in comparison to what I am trying to mimic, is that Python is an interpreted language. This makes running it significantly slower, compared to a compilable language like C. However, this problem is partially mitigatable, which I will touch on later.

Among stand-alone deep learning frameworks for Python there exists two mastodons: Tensorflow and PyTorch. The first is actively developed by Google, the other by Facebook[10]. Tensorflow has been longer in the game, and might have a larger traction. PyTorch on the other hand offers a more python-esque way of writing code, and has a dynamic computational graph, which allows for loops and conditional code inside the network model. PyTorch has also been shown, in some tests, to be able to perform both forward and backward propagation faster than Tensorflow[14]. As a result I chose PyTorch as the deep learning framework to be used in the implementation.

4.2. Implementation strategy

In order to be able to answer the question from the problem definition in a scientific manner, I need to test how various configurations and parameter tweakings change the relationship between learned skill and the required computational effort.

I have decided that a game complexity of the size of Go on a 19x19 board would be

too ambitious. It would still, however, be interesting to investigate how changing the complexity of the game being played affects the learned skill and needed processing power, measured by running time. I have settled on training agent models on board shapes 7x7 and 9x9.

To safely test the various configuration of the algorithm, the code contains a *config* object. This is created when the algorithm is started, and is used extensively when the algorithm runs. Having algorithm parameters and boolean control variables gathered at one place, made it easy to keep track of changes. Several of the variables can be set directly from the terminal, when starting training of an agent model. Doing it this way, I avoided having to directly change the code for some part, if I wanted to test something else. For instance, in the implementation of the network, a variable is consulted in whether forward propagation should be following the original residual block design[1] or full pre-activation [2].

4.3. Optimization

To generate enough trained agents, and play them against each other, inside the time scope of this project, an optimized code base was needed. As mentioned earlier, Python is an interpreted language which affects how fast it runs. Additionally, Python, as implemented by CPython, is not really suited for multi-threading, as it employs what is know as the Global Interpreter Lock (GIL)[9].

4.3.1. Numba

To get around the slowness in Python, resulting from the fact that it is interpreted, I use the just-in-time compiler *numba*. Numba translates part of the code into machine code, which enables it to run much faster. To actually gain a performance boost from this, it should not be used on Python objects, as the GIL will then hamper execution speed. However, both regular primitive values, and most python and numpy collections, of fixed length, can be utilized. Numba especially improves performance of code that run loops.

4.3.2. Playing and training in batches

I decided against trying to parallelize the code running on the CPU with either threads or multiprocessing, because of the GIL. Instead, to improve the running performance, I made use of PyTorch built-in functionality to make neural networks work on batches of

[1] See figure 3.1.
[2] See figure 3.2.

data, thus utilizing parallel data processing on the GPU. This improved the running performance of the algorithm tremendously, allowing self-play game data to be generated in less time, and training on said data much faster.

4.3.3. Half-precision

A final optimization that was done was to use half-precision floating points. As touched on previously, the Nvidia RTX GPUs contains Tensor cores. To use these, the data needs to be half-precision. Initially I tried doing the neural networks and all input in half-precision, but at some point I experienced a network where all the weights of a particular network ended up being *NaN*. Without being completely certain, this probably happened because some accumulator value became too large to represent in binary16, thus becoming ∞, spread this value to other variables, and finally ended up with diving two infinite numbers, resulting in *NaN*.

The final code contains the option to use mixed precision and single precision. With mixed precision, a single precision master version of the network is kept at all time. It is this model that is trained, on single precision input. I did not feel confident enough to attempt the exact same mixed-precision training techniques described by Micikevicius et al.: weight scaling to avoid vanishing gradients, and only converting accumulators to single precision[18]. However, under my version of mixed precision, all self-play games are played with a copy of the network, where all network layers, except for Batch Normalization which does accumulation, are converted to half-precision.

A last place where single precision is used, and at all times, is when saving algorithm data to disk. Here all game data is converted to numpy's implemenation of the binary16 datatype, float16. Before I did this, game data values were saved as regular Python floats, which are objects that take up 64 bits each. The result being that disk space used by game data was reduced by a factor of ≈ 4.

4.4. Hex

The game of Hex was implemented in the simplest possible way. It contains several functions that are used by the algorithm: *initial_state(size)*, *actions(state)*, *player(state)*, *utility(state, player)*, *terminal_test(state)*, and *result(state, action)*. The functions return exactly what their names imply.

As can be seen, at the core of all these functions is a *state*. A state has been modelled as a class with information relevant to these functions. Among the most important variables is the board, conceptualized as a 2-dimensional board[3], where 0 represents an unoccupied board position, 1 occupied by the first player, and -1 is occupied by

[3]But in the code actually modelled as a 1-dimensional list.

the second player. Using a conceptually 2D board as a hexagonal grid is possible when certain restrictions are applied on the possible directions of connection. An illustration of this can be seen in figure 4.1.

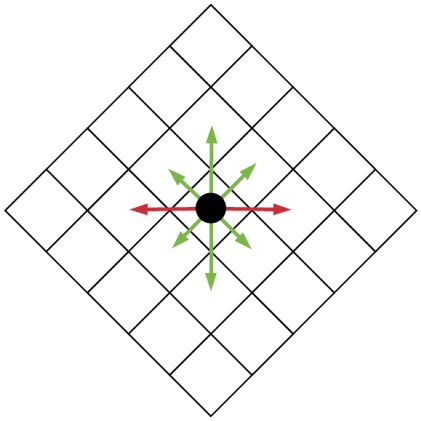

Figure 4.1: Example of using a 2D array as a hexagonal grid representation, by only allowing connections between the cells with a green line, and disallowing the ones marked with a red line. The result is that only 6 connections are possible, making the board practically hexagonal.

A state also includes a list of actions taken previously, in the form an list of integers is kept, for quickly creating input images for the neural network. Available actions are stored as a set, to quickly remove an element when an action is taken. Finally it also includes an array used in the Quick-Find variant of the Union-Find algorithm by Sedgewick and Wayne[21]. This is used to quickly determine if a game state is terminal, i.e. a player has connected its two sides. This array contains an entry for each positions on the board, and also 4 entries representing the sides of the board. That way it is easy to keep track of what is connected, in regards to the rules of Hex.

To further minimize the computational footprint of the game implementation, almost all functions are fully or partially compiled by numba during run time. The result is that the game itself is the least concern in regards to the running performance of the algorithm.

4.5. The learning agent algorithm

The algorithm has been implemented according to the outline presented in figure 3.5. In the following I will delve a bit more into the details. What is not seen in that figure, is that in my implementation games are played in batches of 50. Furthermore, between each batch run of self-play games, the network trains for 10 iterations. What was needed to be balanced was processing speed, but also getting sufficient data for the algorithm, and for the network, not to train too much or train too little. The latter is important early on when the agent who played the games are not very smart, and we want to avoid overfitting. The numbers 50 and 10 were found after trying different sizes of varying success, with these showing good results, when I examined their training loss rates.

Because of the limitation on processing power the number of MCTS simulations was set to 200. This value was kept for both on 7x7 and 9x9 boards. AlphaZero used 800 simulations for all its games. For Go on a 19x19 board, that amounted to a mean of ≈ 2.22 visits for each child node of the initial state. For Hex, on 7x7 and 9x9 boards, it amounts to ≈ 4.08 and ≈ 2.47 respectively. I found it to be a reasonable sample sizes, when keeping in mind what AlphaZero employed for Go with its larger complexity.

$$UCB(p, c) = \frac{V_c}{N_c} + 1.27 \times P_c \times \sqrt{\frac{N_p}{1 + N_c}} \tag{4.1}$$

The balance between exploration and exploitation, in the MCTS part of my algorithm, is done by the version of UCB seen in equation 4.1. This is very similar to equation 3.3 and 3.4 from the previous chapter, with the only difference that $C(p)$ has been replaced by an exploration constant of 1.27. Had this constant been higher, more states would have been visited, potentially helping avoid a horizon effect, but at the cost of being less certain about the most promising actions. When the maximum number of MCTS simulations has been reached, and an action selection is performed by doing ArgMax on the root's child nodes' visit counts.

4.5.1. Mitigating horizon effect

The probabilistic random selection of actions, mentioned earlier, has also been implemented for an initial number of steps during each training game. This is set to a game's maximum length divided by 10, and ceiled. For a 7x7 board it means that this form of selection is active for the first 5 moves of the game, and for 9x9 the first 9. The probabilistic values are created with values fed to a Softmax function. The values are the normalized visit counts for the root's child nodes. The normalization is to avoid less frequently visited nodes gaining a probability of practically zero, if the distribution is very skewed. Having all the values be below 1 flattens the distribution curve, elevates the less visited nodes' probability, and lowers that of the most visited. To prevent it becoming too flat, all values are multiplied by 1.5 before Softmax.

Dirichlet noise is also being added to the prior values, with the network's prior predictions counting for $3/4$, and the normalized dirichlet values counting for $1/4$. In the config object, the dirichlet α-value is set to 10 divided by the average number of legal moves of the game. This I have estimated, for Hex, to be the number of board positions multiplied by 0.7. In retrospect, this estimation is put way too high, especially for games where the playing agents know what they are doing. But from experience, the number seemed to work fine in creating suitable α-values, and was automatically adjusted when the board dimensions were changed. Sorted examples of how the noise distributions can look, after they have been normalized and multiplied a factor of 0.25, can be seen in figure 4.2.

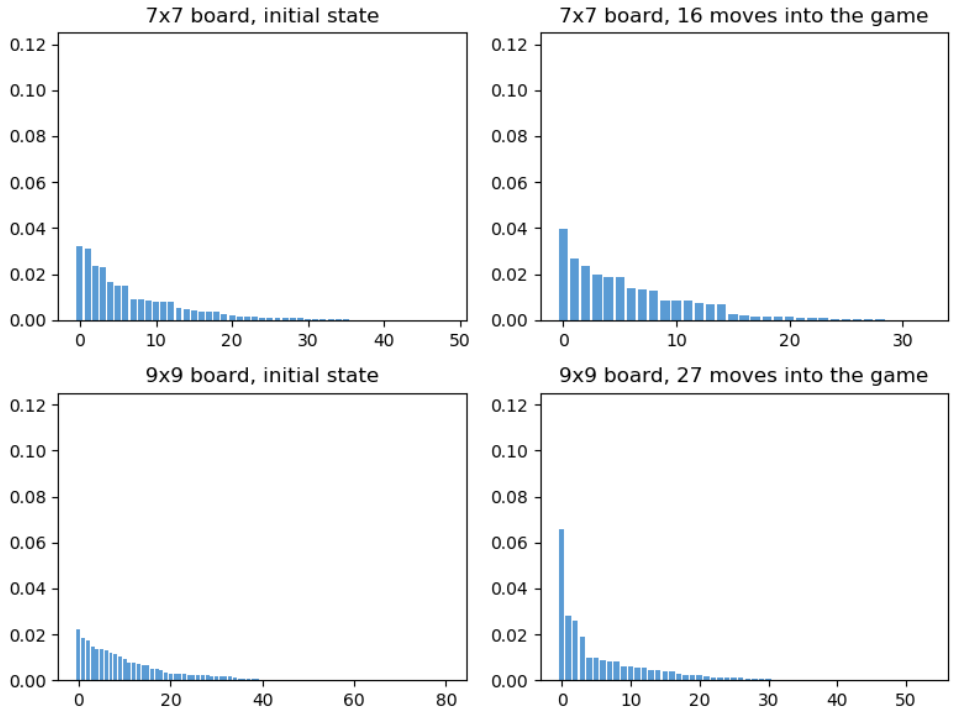

Figure 4.2: Example of 4 different sorted dirichlet noise distributions, after values have been normalized to a sum of 1, and then multiplied by a factor 0.25.

4.5.2. Network structure

The neural network is set to build residual blocks with convolutional layers of channels size 256 and stride 1. Earlier attempts at setting the number of channels to 128 greatly improved speed, and the networks size on disk, but it hampered accuracy enough for me to raise it. The network can have its number of residual blocks adjusted. I ended up testing residual block counts of 13 and 19, to see how much impact it has having it lower than AlphaZero's 19. Furthermore, it can also be tasked with doing forward propagation either in accordance with the original residual block design[4] or the full pre-activation design[5][11, 12]. Otherwise it is built pretty much as described by Silver et al., and shown in figure 3.3.

4.5.3. Training

Training is done with a Stochastic Gradient Descent optimizer that uses a momentum[6] of 0.9 , weight decay[7] of 0.0001, and a learning rate of 0.05. The first two are similar to the ones used by AlphaZero, but the last is 4 times lower. This was found by brute forcing, as my implementation resulted in loss rates that would be all over the place when it was higher. Unlike AlphaZero the learning rate is not lowered as the algorithm progresses for several numbers of games being played.

[4]See figure 3.1 for an illustration.
[5]See figure 3.2 for an illustration.
[6]A technique used to speed up Gradient Descent convergence.
[7]Used for regularization of weights, to prevent overfitting

The loss functions are Mean Squared Error for values, seen in equation 4.2, and Softmax Cross-Entropy with Logits for policies, seen in equation 4.3. For equation 4.3 the multiplication in $Log(Softmax(\hat{Y}_i)) \times Y_i$ is an element-wise multiplication, not a matrix multiplication.

$$MSE = \frac{1}{n} \times \sum_{i=1}^{n} \left(Y_i - \hat{Y}_i \right)^2 \tag{4.2}$$

$$SoftmaxCrossEntropyLogits = \frac{1}{n} \times - \sum_{i=1}^{n} \left(Log(Softmax(\hat{Y}_i)) \times Y_i \right) \tag{4.3}$$

4.5.4. Network input

Providing the network information about a board state is, everything considered, domain-specific knowledge. However, I have strived to make the features included as minimal as possible.

Input is provided to the network as images of features, in the form of PyTorch tensors[8]. For a 9x9 board it has the shape of $batchsize \times 2 \times 9 \times 9$. Here the number 2 is the amount of channels. I have reserved the first channel for a binary representation of where the player whose turn it is, has pieces on the board. The other channel is for the opponent's pieces.

As such, the features given to the network is very minimal. It is not given information about how connection restrictions are applied to use the 2D board array as a hexagonal grid, and it is not given information about which realized connections exist for either player. These elements is up to the networks to learn themselves.

One trick on the input, to ease the learning, is applied. That is, if it is the second player's turn, both channel board images are transposed, such that the perspective emulate that of the first player, in terms of the turn taking player's goal sides, and which belongs to the opponent.

4.5.5. Other parameters

For each 5 of the play-50-games-train-network-10-times iterations, everything of interest is saved to disk. This includes the network, the config object, the 1,000 or 500 most recent games, the loss values recorded during training, the time spent running, etc.. This makes it possible to stop the algorithm, for whatever reason, and restarting it again from the last save, or use it for normal play mode.

[8] A PyTorch tensor is basically a matrix array.

5. Experiments

All experiments have been conducted with some of the parameters being consistent in all the experiments. These include all convolutional layers of the residual blocks having a channel number of 256 and a stride of 1. The networks all have a learning rate of 0.05, and 50 games have been played between training, and training is done with 10 training iterations between game-playing. All decisions during game play have been found through 200 MCTS simulations, with the neural network providing evaluation and guiding the search. Furthermore, all experiments have run until 1,200 training iterations have been reached, thus 6,000 games of self-play has been generated for each experiment.

Some variations have been made to measure the effect of parameter tweaking. I have created 4 main categories of experiments: (A) is the one that models AlphaZero the most, by using the original residual block design, *utility* of the game result as value training target, and only asking the network in the *evaluate* function in tree search; (B) is similar to A, except it uses *utility* function for terminal states in *evaluate*; (C) further extends B by using the found mean value of game states, during tree search, as value training targets; and finally, (D) extends C by using the full pre-activation design for its residual blocks.

5.1. 7x7 board

The 1st generation of experiments I did on 7x7 boards all used half-precision both during training and self-play. All of them also used 19 residual blocks. The experiments have been labelled with a 1 appended to their corresponding category. This is listed in table 5.1.

After I encountered a *NaN* issue while training a network in half-precision on a 9x9 board, I implemented mixed precision. Here training is done on a single precision master network, and self-play is done on a half-precision copy of the master network. All the 2nd generation experiments on 7x7 boards were done with mixed precision, and also on networks with only 13 residual blocks, but with a larger size of former games saved, 1,000 instead of only 500. Like with the 1st generations, all experiments has been labelled by category, and now with a 2 appended. A final 3rd generation experi-

Version name	*A1*	*A2*	*B1*	*B2*	*C1*	*C2*	*D1*	*D2*	*D3*
Games in storage	500	1,000	500	1,000	500	1,000	500	1,000	1,000
Residual blocks	19	13	19	13	19	13	19	13	19
Precision	half	mixed	half	mixed	half	mixed	half	mixed	mixed
Utility in evaluate	false	false	true	true	true	true	true	true	true
Value training target	utility	utility	utility	utility	mean value	mean value	mean value	mean value	mean value
Full pre-activation	false	false	false	false	false	false	true	true	true
Running time	9h 20m	8h 19m	7h 49m	6h 45m	6h 21m	6h 16m	8h 3m	6h 29m	12h 12m

Table 5.1: List of experiments for 7x7 boards, all with 1200 training iterations and 6000 self-play games.

ment was also done, but only for the D category, which modelled the 2nd generation, but with 19 residual blocks. These can all be seen in table 5.1.

It should be mentioned that the running time numbers from table 5.1 should be taken with a grain of salt, since the hardware the training has been performed on has not been dedicated exclusively to the experiments, i.e. other applications might have been running at the same time. Still, a tendency can be observed, that suggests: fewer residual blocks lead to faster training.

5.1.1. Training losses

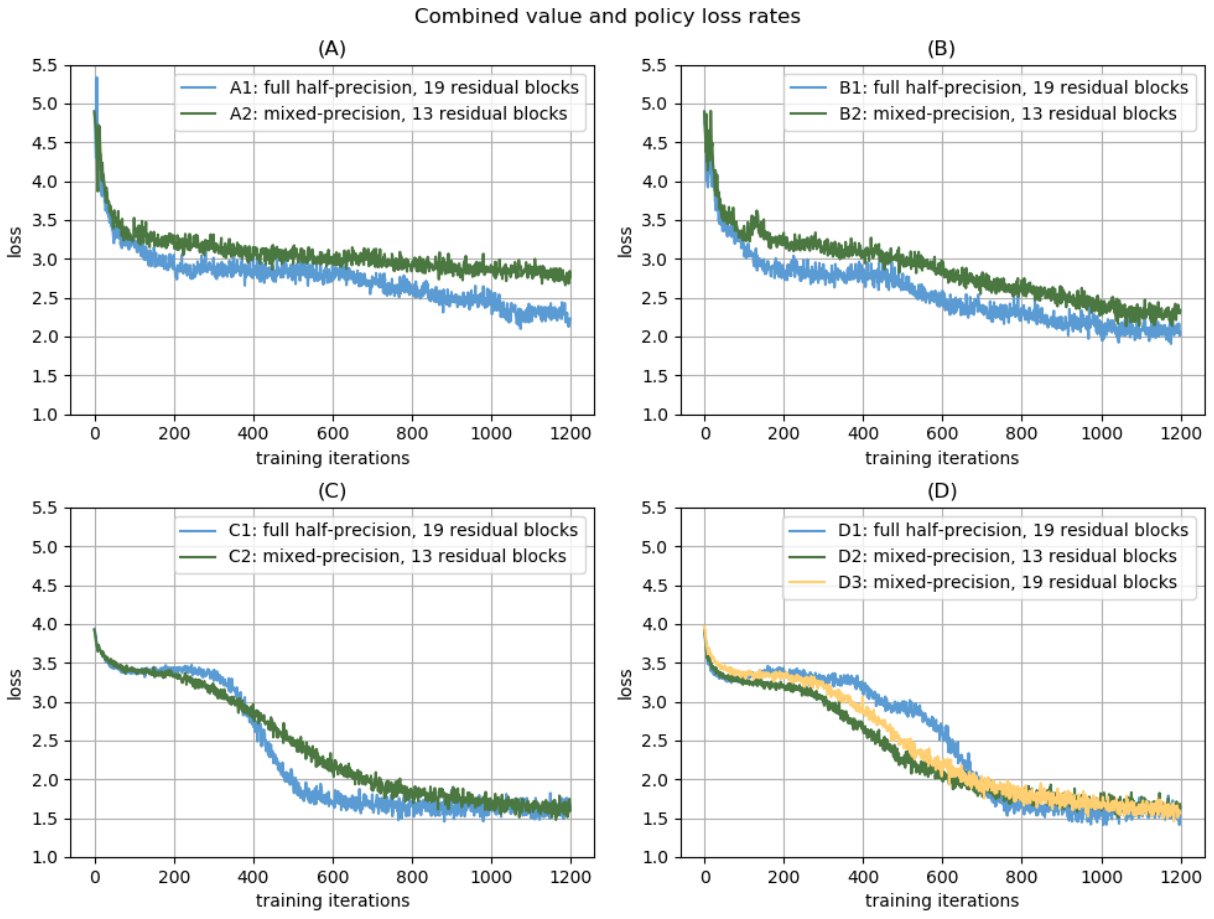

Figure 5.1: Combined value and policy training loss rates for various models, trained for a 7x7 board.

If we take a look at the combined training loss rates for the various experiments, in figure 5.1, we can see that precision type barely affects the curves. It is unfair to compare the curves of A and B, in terms of their distance to 0, with the ones from C and D, since they don't use the same value training target.

Comparing A to B, we can observe that both of the B experiments end up with a loss rate below 2.5, whereas the shallower mixed precision experiment A2 ends above 2.5. C and D almost have the same curves for all experiments, with the outliers being C1, which however, ends up around the same area as the others. Another interesting observation is that both of the 2nd generation experiments of A and B have curves above their 1st generation counterparts, whereas for C and D this is not the case. This indicates that either having fewer residual blocks, or having to perform single to half precision, is having a negative impact when the value training target is the utility of the game result.

5.1.2. Win rates against MCTS

Training loss rates illustrate how the neural networks perform in predicting on the training data, over time. However, they do no necessarily say anything directly about performance in playing the game. Therefore the same experiments have also been playing against the baseline MCTS agent opponent that used 2,000 simulations. For every 50 training iteration, experiments have played 20 games against this opponent, 10 as the starting player, 10 as the second player. The results can be seen in figure 5.2.

Even though the amount of games played for each data point in the lines in figure 5.2 are relatively few, the graphs still suggest several patterns. Both C and D, which use mean values as value training targets, reach situations where they more or less confidently beats their opponent, around 500 training iterations. Additionally for C and D, there does not seem to be much difference, whether the neural networks has 13 or 19 residual blocks, use the original residual block design or full pre-activation, or is in fully half-precision or mixed precision.

For A and B, we can observe that feeding the algorithm real game information by the utility function, for terminal states, seems to have a noticeable effect, as the B experiments reach high win rates earlier. Unlike C and D, both of the 2nd generation A and B experiments, A2 and B2, are slower at reaching the win rates of their 1st generation counter parts. Especially A2 never reaches any impressive playing performance.

The disparity of win rates between the generations of A and B interestingly coincides with the 2nd generation experiments also having higher loss rates. At the same time, for C and D, no experiments have distinct disparity in either win rates or training loss. This suggests that either the fewer residual blocks or the numerical precision conversions is harmful when using the utility value of the game result at value training target. At the same it does not seem to be harmful when using mean value as value training target.

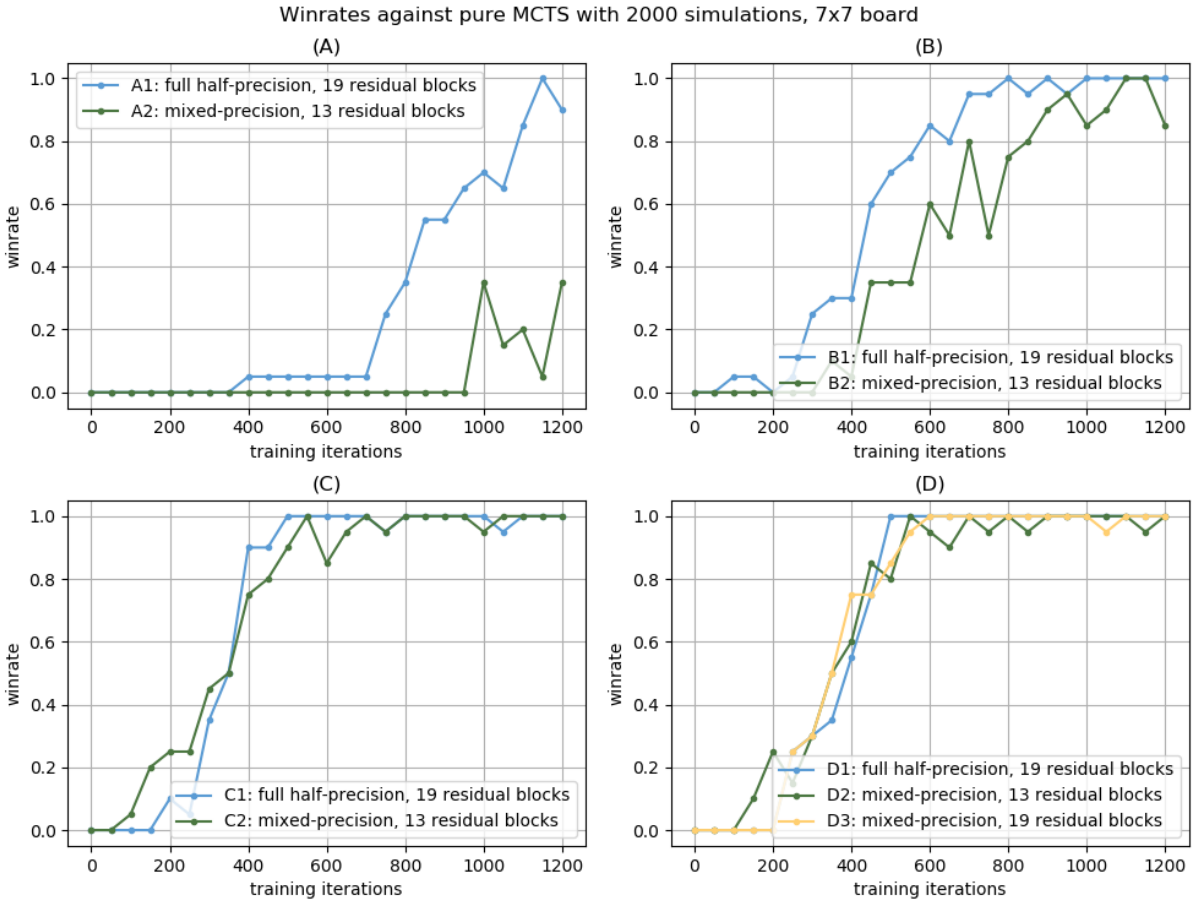

Figure 5.2: Winrates of various models, playing on a 7x7 board, against a purely MCTS implemented opponents, running with 2,000 simulations. Each data point is win rate from 20 games, where the learning agent model has played half as first player, and the other half as second player.

5.2. 9x9 board

For the board size of 9x9, the same experiment category definitions as for 7x7 are used, with a mapping of (E) = A, (F) = B, (G) = C, and (H) = D. For the 9x9 agent experiments, only a single generation of experiments have been done, except for H. The 2nd of H, H2, is the only agent among all experiments that has used single precision both for training and self-play. All 9x9 agents have employed neural networks 19 residual blocks deep. All 1st generation experiments have been done with mixed precision, because a failed experiment at fully half-precision resulted in all weights of the network becoming *NaN*. A listing of the experiments can be seen in table 5.2.

As with the 7x7 experiments, these have been running for 1,200 training iterations, generating 6,000 self-play games. All experiments have retained the last 1,000 played games to sample training data from. Thus, even though the game complexity has been considerably increased, all the 1st generation 9x9 experiments are very similar in set-up to the 2nd generation 7x7 experiments.

Again, the running times are potentially misleading, as the hardware may have been running other processes simultaneously to the experiments. If they are to be believed, they could indicate that the G1 and H1 experiments learn more optimal plays, thus re-

Version name	*E1*	*F1*	*G1*	*H1*	*H2*
Games in storage	1,000	1,000	1,000	1,000	1,000
Residual blocks	19	19	19	19	19
Precision	mixed	mixed	mixed	mixed	single
Utility in evaluate	false	true	true	true	true
Value training target	utility	utility	mean value	mean value	mean value
Full pre-activation	false	false	false	true	true
Running time	21h 38m	19h 56m	15h 58m	14h 41m	1d 1h 17m

Table 5.2: List of experiments for 9x9 boards, all with 1200 training iterations and 6000 self-play games.

ducing the average game length, consequently reducing the required processing power to reach 6,000 self-played games. The outlier among all the 9x9 experiments is H2, which ran for more than a day.

5.2.1. Training losses

Looking at the training loss rates, in figure 5.3, once again the experiments using mean value as the value training target, G and H, are experiencing greater training losses over time. This time, however, the second drop in rates occurs later than for the C and D experiments.

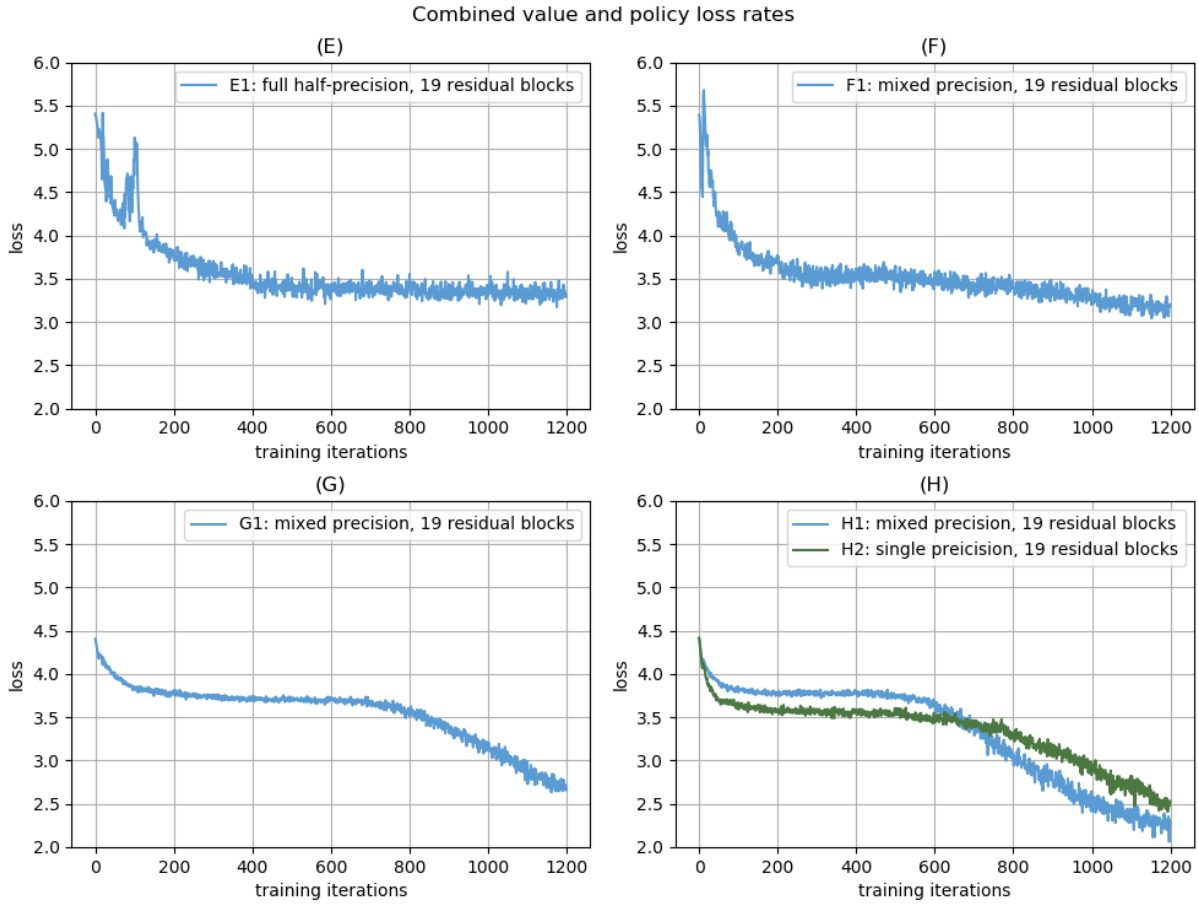

Figure 5.3: Combined value and policy training loss rates for various models, trained for a 9x9 board.

Interestingly H experiences a deeper drop than G, which also starts earlier. Among the two H experiments, H2 experiences a lower loss rate until around training iteration 700, when H1 makes a deep drop, which ends up having the lowest by the 1,200 training iteration mark. E1 and F1 follow the same slowly decreasing trend as their 7x7 mixed precision counterparts, A2 and B2.

5.2.2. Win rates against MCTS

As mentioned in chapter 2, an increase in board size from 7x7 to 9x9 greatly increases the game tree complexity. Using an MCTS opponent with only 2,000 simulations on a 9x9 board would make it significantly intellectually impaired, in contrast to when it plays on a 7x7 board. Therefore I have increased the number of simulations for the baseline MCTS agent to 10,000. However, this also increases the computation time tremendously, because the agent is not utilizing parallelism. As a consequence only 10 games have been played between the baseline MCTS and each experiment, at 50 training iterations interval, with 5 as first and 5 as second player. The resulting win rates are found in figure 5.4.

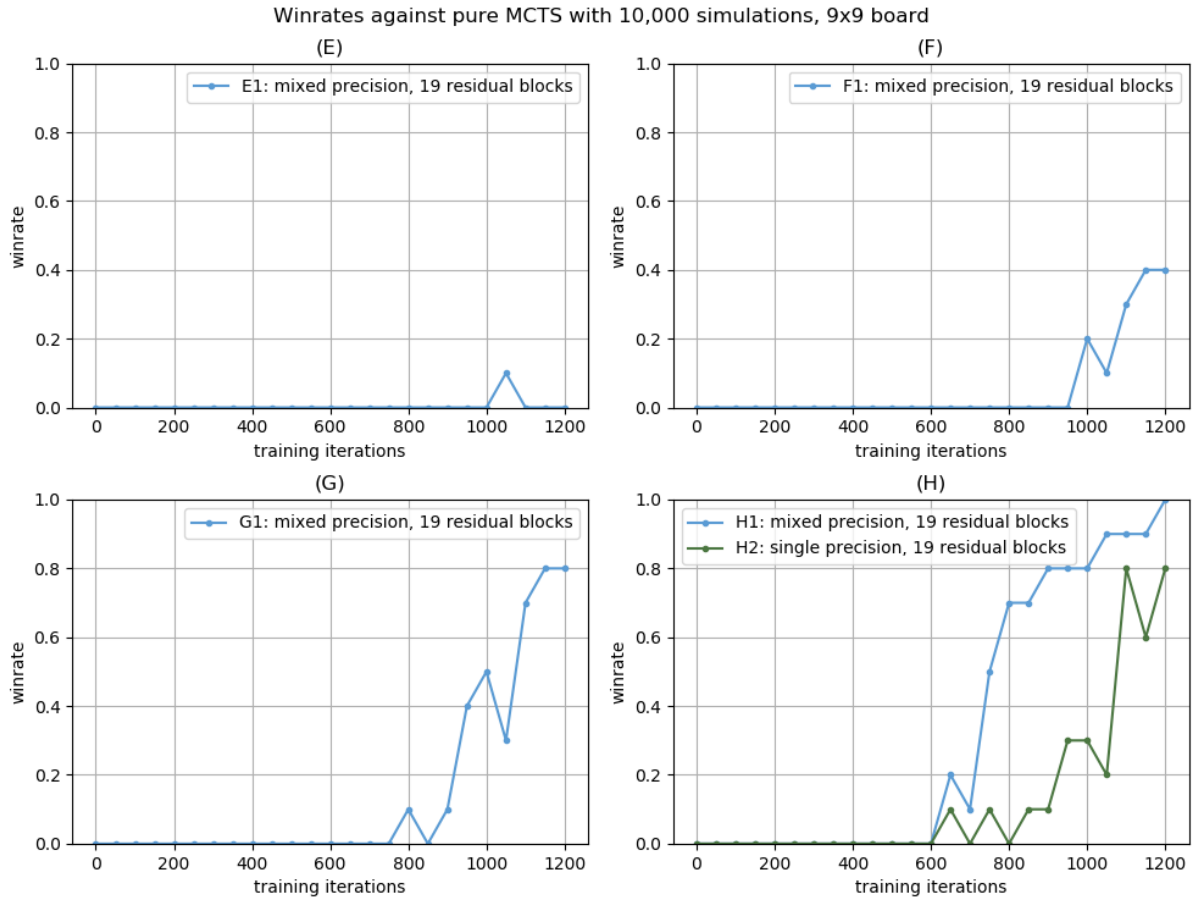

Figure 5.4: Win rates of various models, playing on a 9x9 board, against the baseline MCTS implemented opponent, running with 10,000 simulations. Each data point is win rate from 10 games, where the learning agent model has played half as first player, and the other half as second player.

Like with the 7x7 experiments, win rate follows training loss to a large degree, with the experiments that experienced the largest drops, also being the ones that play better.

Different from 7x7, full pre-activation actually seems to perform better than the original residual block design, in that H1 has a better win rate than G1. Surprisingly E1 loses all its games, except for a single one. F1 shows almost the same low level of playing ability, but does have an up-going curve towards the end of the experiment. A final interesting observation is that the single precision version of H, H2, fares significantly worse than its mixed precision counterpart.

5.2.3. Performance against Hexy

In order to test the game playing performance against an established well-performing agent, I have conducted tests against Hexy, the agent mentioned in chapter 2. The games have been played in a round-robin tournament format[1]. Here H1, H2, the baseline MCTS, and Hexy have all played against each other. Hexy on expert settings, and baseline MCTS with 10,000 simulations. To track the learning curve of H1 and H2, versions from 200 training iteration intervals are included. All match-ups have had the agents alternate between being first and second player, to get around the first player advantage. Furthermore, to add more certainty to the model, all H agents that have played the baseline MCTS, has done so for 10 games, 5 as first and 5 as second. All game results are listed in appendix D.

A freeware tool to estimate bayesian Elo ratings, by French computer scientist Remi Coulom, has been used to measure the playing performance of the agents[7]. The tool is originally created for Chess, but, to the best of my knowledge, equally suitable to the task at hand. The output of running the tool can be found in appendix C, and a graph of the results are in figure 5.5.

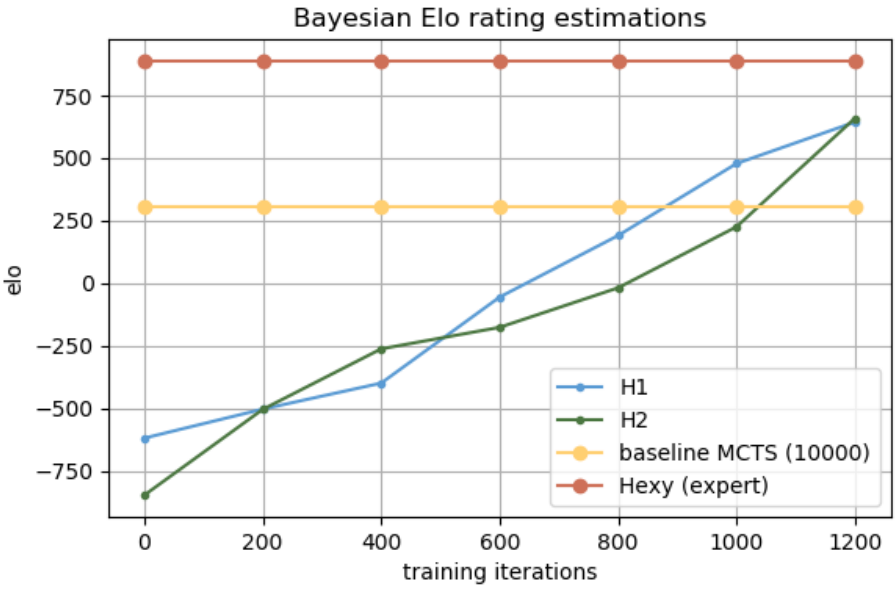

Figure 5.5: Bayesian Elo rating estimations for H1 and H2, over time, and static versions of Hexy on expert settings, and baseline MCTS with 10,000 simulations.

[1]Also known as a all-play-all tournament.

For the games supporting figure 5.5, Hexy did not lose a single game. Interestingly, according to the estimates, H2, actually fares slightly better than H1, for the 1,200 iteration versions. To test if is actually possible to beat Hexy, on expert settings, with an agent using this project's approach, I let H2 train for an additional 1,200 iterations.

H2 with 2,400 training iterations, and 12,000 self-play games played, spent 1 day 22 hours and 10 minutes running in total. Playing it against Hexy, still on expert settings, H2 was able to beat Hexy in the game where H2 was the first player. It lost the second game, where it was the second player. Including these two games in the elo estimation, Hexy and H2 with 2,400 training iterations are estimated to be of equal Elo rank. However, with a much larger uncertainty for H2. The play history of both games can be found in appendix B. The Elo estimation tables are in appendix C.

6. Discussion

The experiments show that the algorithm benefits from being fed more real game information, by using the utility function when reaching a terminal state. Witnessed in reaching a good playing performance earlier. I still see this as being within the general purpose intention behind the algorithm, since *utility* is a generic game function that is also being used for the value training targets in AlphaZero.

It is also clear that having a different value training target than utility of how a game ended is beneficial, at least when it is the tree search mean value that is being used. To do this, however, one has to feed real game information into the algorithm, for instance with the approach mentioned in the above paragraph. One can speculate that having limited discrete end game values does not reflect how a non-terminal game state value is best approximated. The results seem to suggest this.

The full pre-activation residual block design also had an impact on the agents trained on 9x9 boards. The researchers who came up with this design explain that the order of activation functions, whether they occur before or after layers, matters only in networks that have branching in their layers[12]. A residual block has branching in the form of the identity mapping of the input added at the end. Their results showed great performance in classifying ImageNet datasets. I argue that it is also shows a positive effect when used on abstract games, when used to learn value and logits, and input is given in the shape of a image of game features. Interestingly, when trained on a less complex board size, 7x7, it did not matter whether the original block design or the full pre-activation was used. And the same goes for the shallow and deep network variants of the full pre-activation experiments, on the 7x7 board. They performed approximately the same. This indicates that doing activations before weight layers has an impact that increases with the difficulty of the prediction task.

Mixed precision also showed some good results. One could assume that a fully single precision agents, H2, or fully half-precision, A1, B1, C1, and D1, would perform better than ones that has to do conversion from single precision master network to a half-precision one, potentially losing accuracy in the process. However, on the 7x7 experiments it did not seem to matter for the networks that used a mean value as value training target. On 9x9 it even had a positive effect for the full pre-activation experiment. This supports the hypotheses from He et al., which suggests that full pre-activation improves regularization of network models[12], and Micikevicius et al. which suggests

that converting from single to half-precision may act as a regularization[18]. However, to say anything conclusive about this more and longer expirements would have had to be run, since H1 and H2 ended up with about the same playing performance. If it is indeed the case, then the other experiments could have been suffering from overfitting to some degree. One could have tested some other values for weight decay and learning rate, to see if they helped in reducing overfitting. But, if full pre-activation and mixed precision does the job, as the results seem to indicate, it is an easier and less domain-specific approach.

It is hard to draw conclusions on the playing performance when going from one game complexity to another, a 7x7 board to 9x9 board, since the baseline MCTS opponent had to be adapted. What can be observed is that the experiment that most closely emulated the original AlphaZero approach, although severely scaled down, had significantly learning issues, compared to the other experiments. Also, full pre-activation seemed to matter more on more complex problems.

In terms of the agents acquiring a game-playing performance that makes them competitive against state-of-the-art Hex agents, it is clear that 1,200 training iterations have not been sufficient, on the 9x9 board. That said, the findings seem to suggest that doubling the training time, does move the agent in this direction. This is witnessed by H2 with 2,400 training iterations, and 12,000 self-play games played, were able to beat Hexy, set to expert settings. This indicates that 6,000 self-play games is not enough to learn this level of skill, but it is enough, given the right design of the algorithm, to beat the baseline MCTS with 10,000 simulations, quite confidently.

Looking at figure B.5 in appendix B, the play history indicates that H2, with 2,400 training iterations, has actually learned some strong global virtual connectivity patterns. Virtual connections are plays that create yet to be realized connections, where the opponent is unable to prevent a given outcome. In some cases H2 is stronger than Hexy, but it still has shortcomings, as seen by the game in figure B.6.

7. Future work

Had there been more time, it could be interesting to let the models train longer, to measure the impact of conducting more games of self-play and perform more training.

It could also be interesting to do experiments with additional parameter tweakings. These could include how the amount of recently played saved games impact the playing performance. Anthony, Tian, and Barber said that their agents, that are somewhat related to the AlphaZero approach, improved from having access to larger datasets[3].

Another motivation for future work could be how the learning rate affects ability to play. One could also attempt to divide the learning rate by a constant, when the network experiences training loss plateaus, as done by He et al.[12]. When researching this, one should bear in mind that the learning rates for C, D, H, and H, from figure 5.1 and 5.3, experiences plateau-like curve sections, before having a big drop in losses. When implementing a decreasing learning rate it might lead to slower convergence - longer time required to gain the same playing performance.

Doing further tests on what actually happens with prediction accuracy and playing performance, when converting a single precision network to half-precision, is only lightly touched upon in this project. Therefore it would be valuable to measure how much information is being lost, when making the conversion, how it is propagated throughout the algorithm, and how that affects the playing performance of the agent.

Changing the noise levels added during tree search is also a field that would be fascinating to investigate. As well as whether adding noise in other parts of the algorithm helps in learning general patterns for game state evaluation.

In regards to the challenge of only having access to limited processing power, it would be interesting to do additional experiments on how deeper and shallower networks perform. In the same respect, it could also be interesting to see how the trained agents perform, in play mode, when the amount of MCTS simulations are reduced, to increase its decision-making speed.

Finally, it would be interesting to let the experiments E1, F1, G1, and H1, follow suit with H2, and run for extended time, to see how much it improves their playing performances. Since H2 uses single precision both during self-play and while training, it has additional costs in terms of running time, compared to the others.

8. Related work

The experiments described in this thesis are primarily related to the AlphaZero algorithm by DeepMind[23]. All experiments follow a similar algorithmic procedure, but scaled down in terms of games played and training done, to work on consumer grade hardware, in a reasonable time. Likewise, the game complexity of the 7x7 and 9x9 Hex boards are nowhere near that of 19x19 Go. Although the approach is very similar, changes across the different experiments have been made, to investigate the effect of said changes. These are: having a different value training target; a difference residual block design; where game information is fed to the algorithm; and using a mix of single and half precision floating points.

For reinforcement learning agents that also start with tabula rasa knowledge, for the game of Hex, this project is also related to the EXIT algorithm[3]. There are however some noteworthy differences. The EXIT algorithm is more closely related to AlphaGo Zero, than AlphaZero, in that it has separate networks for value and policy. Interestingly, unlike the DeepMind approaches, and similarly to some of my experiments, EXIT also uses estimates from the neural network as value training target. But it does not add dirichlet noise to policies, and it reinitializes the network weights before it trains on new data.

The input provided to the network in EXIT includes more game specific features than in this project. In EXIT, the network is given an image of 6 channels: one for the first player's pieces; one for the opponent's; one for first player's pieces connected to one side; one for first player's pieces connected to the opposing side; one for the second player's pieces connected to a third side; and one for the second player's pieces connected to the last side[3]. The input used in my project is much simpler. Only 2 channels are given to the network, one for the pieces of each player. In this regard, my algorithm gets less domain-specific knowledge compared to EXIT. Finally, EXIT uses a significantly higher amount of MCTS simulations in the tree search, 10,000, as compared to the 200 in this project.

9. Conclusion

In the introduction the following research question was asked:

> *Is it possible for an AI agent, based on the described approach and constraints, to gain a skill-level making it competitive against state-of-the-art game-specific AI agents for the game Hex?*

I have shown that it is possible, inspired by an AlphaZero approach, to teach an agent, decent game-playing skills in the game of Hex, on board sizes of 7x7 and 9x9, starting from tabula rasa. This is witnessed in the win rates against the baseline MCTS agent.

Furthermore, I have laid out how different variants of the original approach improve playing performance, and as a result also lowers the required processing power needed to learn. These variants include: feeding game information to the algorithm slightly differently; changing the value training target; using full pre-activation residual block design instead of the original; and applying a mixed precision approach, which combines single and half-precision.

Finally, I have shown that with extended training it is possible to train an agent that is competitive against a state-of-the-art, anno 2000, game-specific AI agent for Hex.

Bibliography

[1] Vish Ishaya Abrams. *Lessons From AlphaZero (part 4): Improving the Training TargetNo Title*. 2018. URL: https://medium.com/oracledevs/lessons-from-alphazero-part-4-improving-the-training-target-6efba2e71628 (visited on 05/10/2019).

[2] Vadim V Anshelevich. "The Game of Hex : An Automatic Theorem Proving Approach to Game Programming". In: *Scientific American* (2000).

[3] Thomas Anthony, Zheng Tian, and David Barber. "Thinking Fast and Slow with Deep Learning and Tree Search". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon Garnett et al. Curran Associates, Inc., 2017, pp. 5360–5370. URL: http://papers.nips.cc/paper/7120-thinking-fast-and-slow-with-deep-learning-and-tree-search.pdf.

[4] Murray Campbell, A. Joseph Hoane, and Feng Hsiung Hsu. "Deep Blue". In: *Artificial Intelligence* (2002). ISSN: 00043702. DOI: 10.1016/S0004-3702(01)00129-1.

[5] Guillaume Chaslot et al. "Monte-carlo tree search: A New Framework for Game AI1". In: *Belgian/Netherlands Artificial Intelligence Conference*. 2008.

[6] Microprocessor Standards Committee. "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 7542008* (2008). ISSN: 01419331. DOI: 10.1109/IEEESTD.2008.4610935.

[7] Rémi Coulom. *Bayesian Elo Rating*. URL: https://www.remi-coulom.fr/Bayesian-Elo/ (visited on 05/10/2019).

[8] Rémi Coulom. "The Monte-Carlo Revolution in Go [presentation]". In: *Japanese-French Frontiers of Science Symposium*. 2009. URL: https://www.remi-coulom.fr/JFFoS/JFFoS.pdf.

[9] GlobalInterpreterLock. *GlobalInterpreterLock*. 2017. URL: https://wiki.python.org/moin/GlobalInterpreterLock (visited on 05/11/2019).

[10] Jeff Hale. *Which Deep Learning Framework is Growing Fastest?* 2019. URL: https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318 (visited on 05/13/2019).

[11] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 770–778. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[12] Kaiming He et al. “Identity mappings in deep residual networks”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2016. ISBN: 9783319464923. DOI: 10.1007/978-3-319-46493-0_38.

[13] Shih Chieh Huang et al. “MoHex 2.0: A pattern-based MCTS Hex player”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. 2014. ISBN: 9783319091648. DOI: 10.1007/978-3-319-09165-5_6.

[14] Anto John. *Compare two deep learning frameworks: TensorFlow and Pytorch*. 2018. URL: https://developer.ibm.com/blogs/two-famous-deep-learning-frameworks-compared-tensorflow-vs-pytorch/ (visited on 05/11/2019).

[15] Patrick Kennedy. *Case Study on the Google TPU and GDDR5 from Hot Chips 29*. 2017. URL: https://www.servethehome.com/case-study-google-tpu-gddr5-hot-chips-29/ (visited on 05/07/2019).

[16] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: 2006. DOI: 10.1007/11871842_29.

[17] John McCarthy. *The Dartmouth Workshop–as planned and as it happened*. 2006. URL: http://www-formal.stanford.edu/jmc/slides/dartmouth/dartmouth/node1.html (visited on 05/05/2019).

[18] Paulius Micikevicius et al. “Mixed Precision Training”. In: *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*. Vancouver, BC, Canada, 2018. arXiv: 1710.03740. URL: http://arxiv.org/abs/1710.03740.

[19] NVIDIA Corporation. *NVIDIA Turing GPU Architecture*. 2018. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 05/08/2019).

[20] Stuart Russell and Peter Norvig. “Introduction”. In: *Artificial Intelligence - A Modern Approach (3rd ed)*. Pearson Education, Inc, 2010. Chap. 1, pp. 1–29. ISBN: 0136042597.

[21] Robert Sedgewick and Kevin Wayne. “Fundamentals”. In: *Algorithms, Fourth Edition*. Addison-Wesley, 2011. Chap. 1. ISBN: 9780321573513. DOI: 10.1002/1521-3773(20010316)40:6<9823::AID-ANIE9823>3.3.CO;2-C.

[22] Claude E. Shannon. “Computers and Automata”. In: *Proceedings of the IRE* (1953). ISSN: 00968390. DOI: 10.1109/JRPROC.1953.274273.

[23] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* (2018). ISSN: 0036-8075. DOI: 10.1126/science.aar6404.

[24] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* (2017). ISSN: 14764687. DOI: 10.1038/nature24270.

[25] Tom M. Mitchell. “Introduction”. In: *Machine Learning*. 1st. McGraw-Hill Science, 1997. Chap. 1, pp. 1–19.

Appendices

A. Software user guide

The code is organized in several modules:

- **root**, contains files:
  - *play.py* starts the program in play mode.
  - *train.py* starts the algorithm in training mode.
  - *controller.py* handles playing the game, when the program is in play mode, between two opponents. Conceptually it belongs in module *hex*, but placed here because of how Python handles disk file lookup.
- **nnsc**, with the following files:
  - *nnsc.py*, contains the runnable algorithm, both the when it is in training mode and in play mode.
  - *config.py*, contains a class of the same name, that holds all the important tweakable parameters for the algorithm.
  - *network.py*, contains the neural network.
  - *gamestorage.py*, contains a class that keeps, in accordance with a config value, the most recent self-played games in memory, for use in training the network. It is also tasked with creating data batches of images of said games.
  - *savefile.py*, contains a self-titled class, that is able to save all important objects to disk, such that the algorithm can be stopped at a certain time, and restarted later, with the same network, gamestorage, config, etc.
- **hex**, with the following files:
  - *gui.py*, allows for a visual and, in the case of a human player, interactive representation of a game.
  - *game.py*, contains all the game specific implementation for the game of Hex.
- **ais**, contains *mcts.py*, which is a simple purely UCT based agent, and *random.py*, which is an agent that takes random actions. Both of these have been used for

testing that the learned agents are actually increasing their ability to play the game well.

When play.py is run it requires 5 or 6 ordered arguments. The 2 first are which agents that should play, the 3rd is the board size, the 4th is the amount of games to play, and the 5th is an optional parameter *vis*, which sets the game to be played with a visual interface. Running play from a terminal could look like this:

```
python play.py nn!az mcts!2000 7 2 vis
```

Several, unordered, optional arguments, separated by spaces, can be given to train.py. Except for the paramaters *save=<filename>* and *load=<filename>*, if a specific argument setting is not provided, a default value will be used, in creating a config object. Running train.py from a terminal can be as simple as this:

```
python train.py save=new-test size=13
```

The parameters used for the config object, when using train.py, are:

- *size=<value>*. Sets the board shape as $size \times size$. Default value is 9.
- *sims=<value>*. Sets the number of simulations the MCTS part of the algorithm uses. Default value is 200.
- *blocks=<value>*. Set the number of residual blocks that are used in the neural network. Default value is 19.
- *filters=<value>*. Sets the number of channels in the convolutional layers of the residual blocks, such that the shape of input and output will be $batch\_size \times filtes \times size\ size$. Default value is 256.
- *actors=<value>*. Sets the amount of self-played games to be in played in batch between training the network. Default value is 50.
- *epochs=<value>*. Sets the number of training iterations to be performed between each batch of self-playing. Default value is 10.
- *window=<value>*. Sets the amount of most recet self-played games to be saved in gamestorage. Default value is 1000.
- *batch=<value>*. Sets the size of the image dataset used for the individual training iterations. Default value is 512.
- *singleprecision*. Sets the model to be using single-precision, both during training and self-playing. Default value is that mixed-precision is used.
- *type=<type>*. If none are set the algorithm will work similar to what is shown in figure 3.6, and use the original residual block design, as shown in figure 3.1. The *<type>* can be one of the following:

- *u*. Sets the MCTS part of the algorithm to use the utility function for terminal states. See figure 3.7.
- *q*. Sets the algorithm to use average mean value of root states as training value target, in accordance with figure 3.8.
- *fpa*. Sets the model to follow both *q* and use full pre-activation block design in the neural network, as seen in figure 3.2.

It is recommended to run the program from a console user interface that has extensive support for ANSI, since the program uses prints that manipulate the console output through escape sequences.

The following packages are also needed:

```
pytorch (a GPU supporting CUDA is necessary)
torchvision
pillow
pickle
tk
matplotlib
numpy
numba
```

B. Selected matches against Hexy

Eight games are included here, with history provided by labels on each piece, indicating when in the sequence of play it was placed.

All games feature Hexy on expert settings, a game as first, and one as second, against: H1 with 1200 training iterations; H2 with 1,200 training iterations; H2 with 2400 training iterations; and the baseline MCTS with 10,000 simulations.

This selection has been made with the attitude that it is most interesting to see how the best performing agents are playing, and in regards to MCTS, get a perspective of what type of decisions it makes.

Hexy wins all of them except one, figure B.5. This against H2 with 2,400 training iterations, where Hexy is the second player (dark).

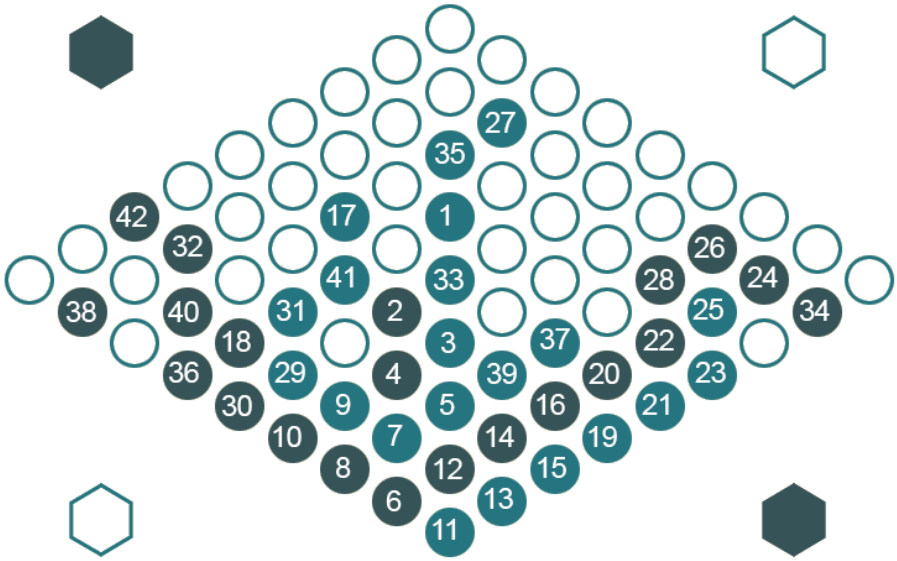

Figure B.1: H1 1200 (light) vs Hexy (dark). Winner: Hexy

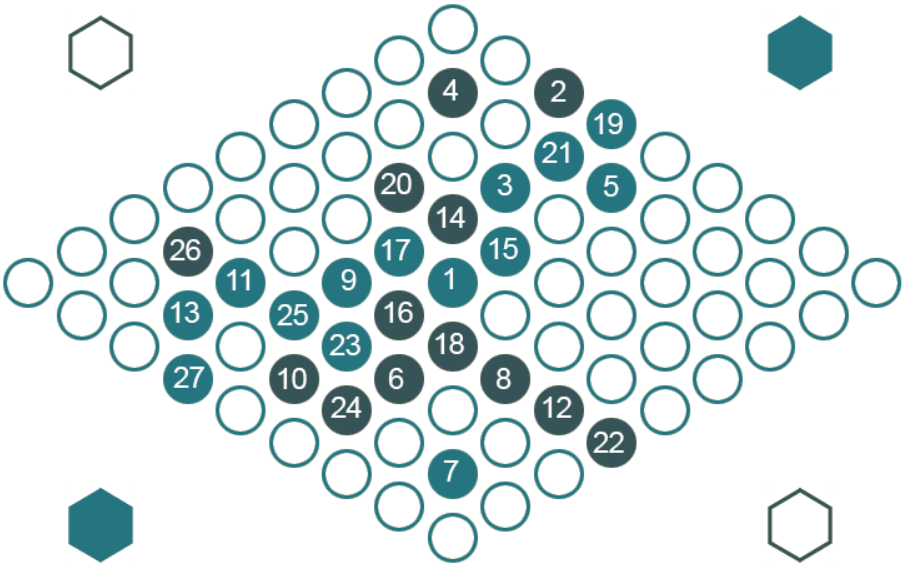

Figure B.2: Hexy (light) vs H1 1200 (dark). Winner: Hexy

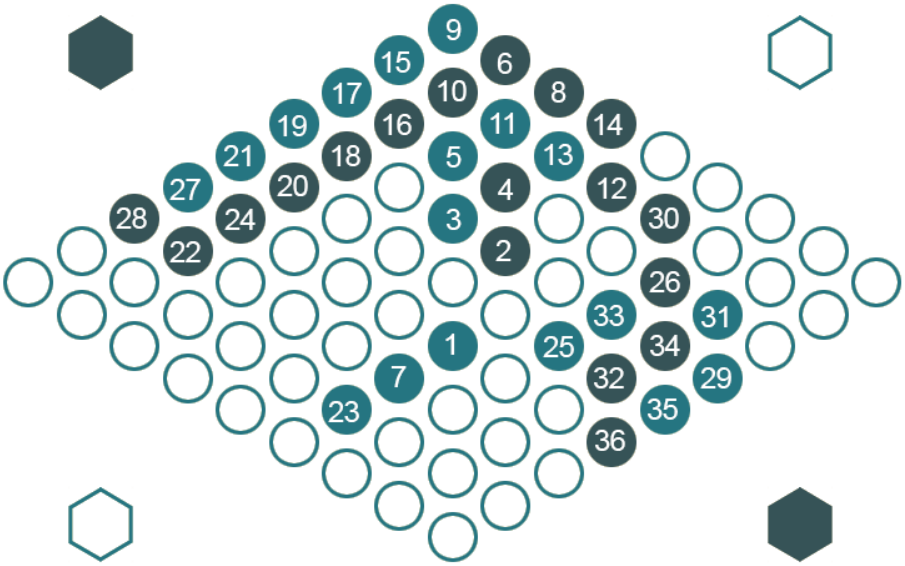

Figure B.3: H2 1200 (light) vs Hexy (dark). Winner: Hexy

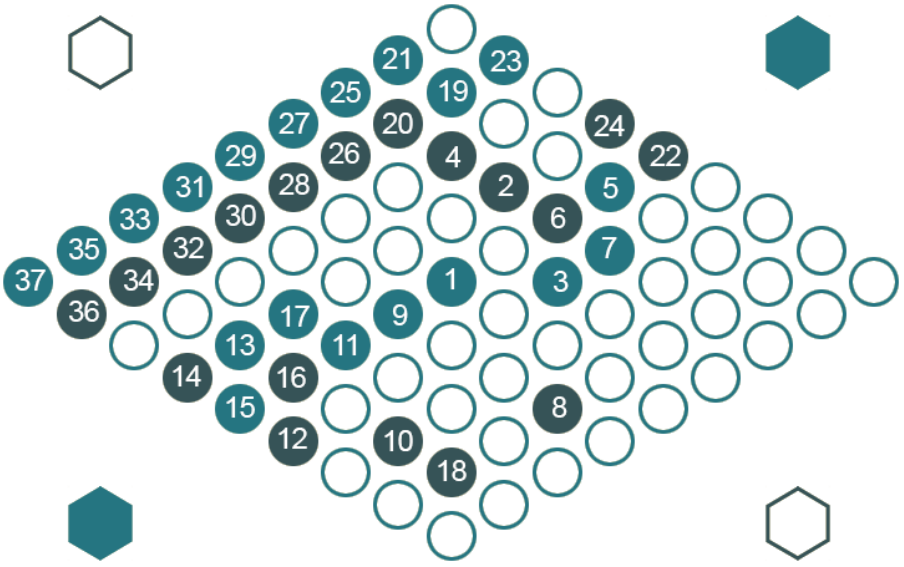

Figure B.4: Hexy (light) vs H2 1200 (dark). Winner: Hexy

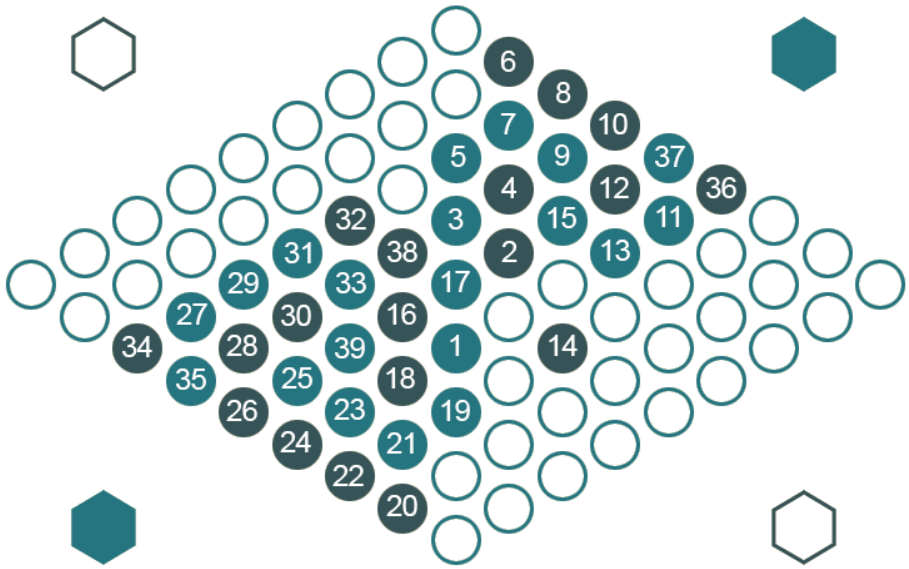

Figure B.5: H2 2400 (light) vs Hexy (dark). Winner: H2 2400

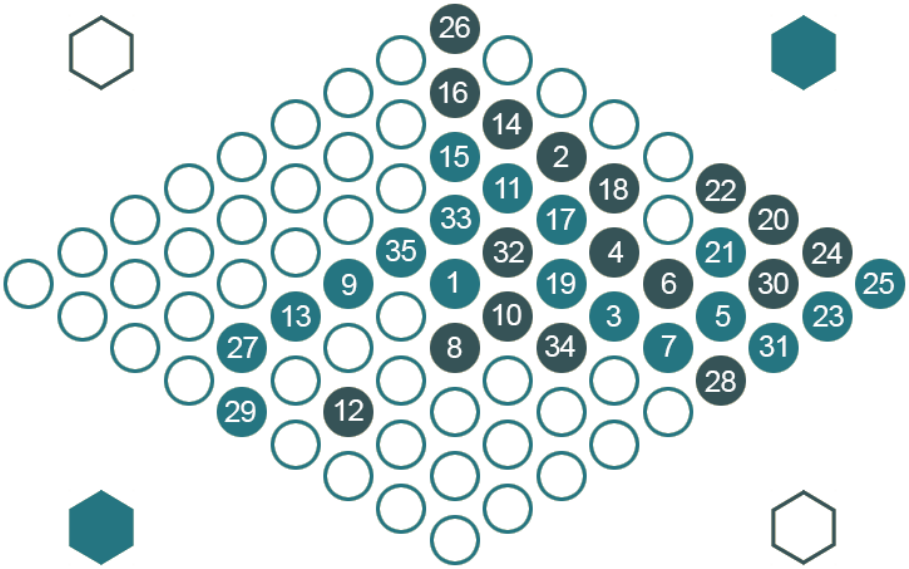

Figure B.6: Hexy (light) vs H2 2400 (dark). Winner: Hexy

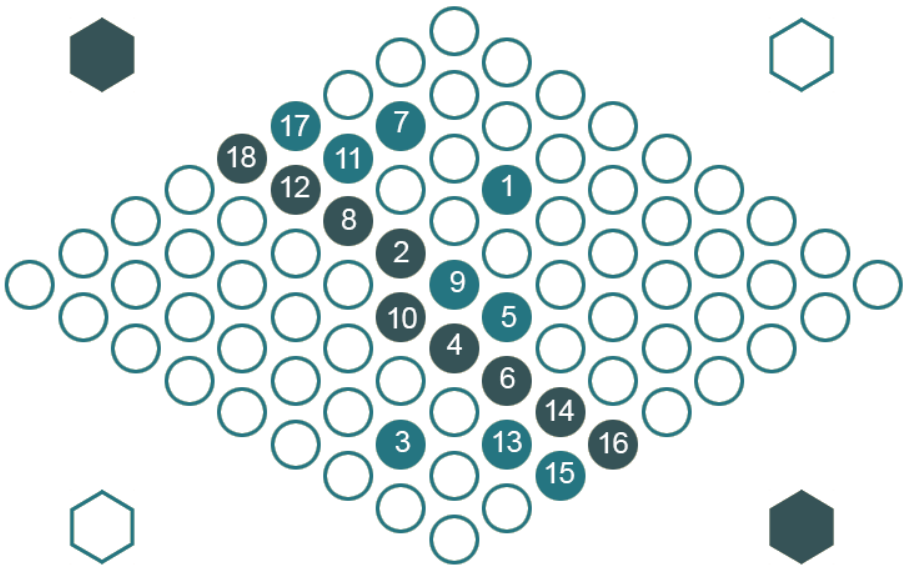

Figure B.7: baseline MCTS (light) vs Hexy (dark). Winner: Hexy

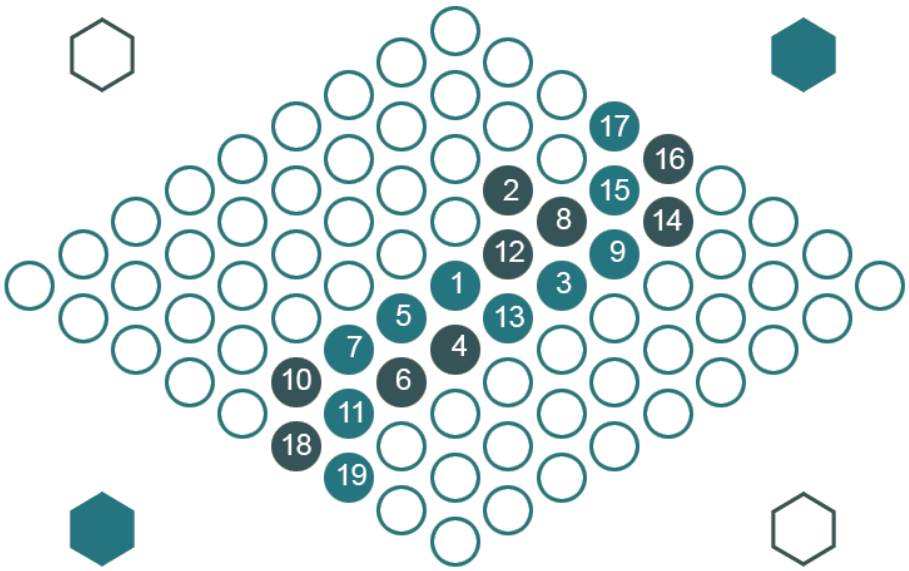

Figure B.8: Hexy (light) vs baseline MCTS (dark). Winner: Hexy

C. Bayesian Elo estimation tables

Below is the output of running the tool [7] from Coulom, with the game results found in Appendix D.

Rank	Name	Elo	+	-	games	score	oppo.	draws
1	Hexy (expert)	888	343	212	30	100%	-59	0%
2	H2 1200	659	183	150	38	89%	30	0%
3	H1 1200	644	179	143	38	89%	31	0%
4	H1 1000	478	145	132	38	79%	40	0%
5	baseline MCTS (10000)	307	79	76	142	75%	-72	0%
6	H2 1000	225	129	130	38	61%	53	0%
7	H1 800	190	131	133	38	58%	55	0%
8	H2 800	-19	131	140	38	42%	66	0%
9	H1 600	-56	132	141	38	39%	68	0%
10	H2 600	-177	138	146	38	32%	74	0%
11	H2 400	-263	143	150	38	26%	79	0%
12	H1 400	-400	150	164	38	18%	86	0%
13	H1 200	-504	158	178	38	13%	91	0%
14	H2 200	-505	161	182	38	13%	91	0%
15	H1 0	-620	167	203	38	8%	97	0%
16	H2 0	-848	199	339	38	0%	109	0%

Adding the two games between Hexy and H2, with 2,400 training iterations, gives the following table:

Rank	Name	Elo	+	-	games	score	oppo.	draws
1	Hexy (expert)	843	229	181	32	97%	-53	0%
2	H2 2400	843	335	342	2	50%	843	0%
3	H2 1200	607	184	151	38	89%	-23	0%
4	H1 1200	593	180	144	38	89%	-22	0%
5	H1 1000	426	146	132	38	79%	-13	0%
6	baseline MCTS (10000)	254	79	76	142	75%	-125	0%
7	H2 1000	172	130	131	38	61%	0	0%
8	H1 800	137	132	134	38	58%	2	0%
9	H2 800	-73	132	141	38	42%	13	0%
10	H1 600	-109	132	142	38	39%	15	0%
11	H2 600	-230	139	147	38	32%	21	0%
12	H2 400	-317	143	151	38	26%	26	0%
13	H1 400	-453	151	165	38	18%	33	0%
14	H1 200	-558	158	179	38	13%	39	0%
15	H2 200	-559	161	183	38	13%	39	0%
16	H1 0	-675	168	204	38	8%	45	0%
17	H2 0	-903	200	333	38	0%	57	0%

D. Game results used in Elo estimation

All results are in Portable Game Notation (PGN), such that it can be parsed by the freeware tool[7] used to estimate Bayesian Elo ratings.

```
[Date "2019-05-13 01:40:16.182317"] [White "H1 0"] [Black "H1 1000"] [Result
"0-1"] .

[Date "2019-05-13 01:35:32.181218"] [White "H1 0"] [Black "H1 1200"] [Result
"0-1"] .

[Date "2019-05-13 02:00:59.765688"] [White "H1 0"] [Black "H1 200"] [Result
"0-1"] .

[Date "2019-05-13 01:52:47.839595"] [White "H1 0"] [Black "H1 400"] [Result
"0-1"] .

[Date "2019-05-13 01:47:33.025522"] [White "H1 0"] [Black "H1 600"] [Result
"0-1"] .

[Date "2019-05-13 01:44:25.662522"] [White "H1 0"] [Black "H1 800"] [Result
"0-1"] .

[Date "2019-05-13 11:18:02.053649"] [White "H1 0"] [Black "H2 0"] [Result "1-0"]
.

[Date "2019-05-13 12:02:12.021380"] [White "H1 0"] [Black "H2 1000"] [Result
"0-1"] .

[Date "2019-05-13 12:05:27.367748"] [White "H1 0"] [Black "H2 1200"] [Result
"0-1"] .

[Date "2019-05-13 11:34:25.461840"] [White "H1 0"] [Black "H2 200"] [Result
"1-0"] .

[Date "2019-05-13 11:40:59.225788"] [White "H1 0"] [Black "H2 400"] [Result
"0-1"] .

[Date "2019-05-13 11:51:38.370201"] [White "H1 0"] [Black "H2 600"] [Result
"0-1"] .

[Date "2019-05-13 11:56:12.637533"] [White "H1 0"] [Black "H2 800"] [Result
"0-1"] .

[Date "2019-05-14 11:19:22.907988"] [White "H1 0"] [Black "Hexy (expert)"]
[Result "0-1"] .

[Date "2019-05-10 20:18:40.588810"] [White "H1 0"] [Black "baseline MCTS (10000)"]
```

[Result "0-1"] .

[Date "2019-05-10 20:18:40.604416"] [White "H1 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 20:18:57.666956"] [White "H1 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 20:19:09.853848"] [White "H1 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 20:19:17.010675"] [White "H1 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 01:42:24.579275"] [White "H1 1000"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 03:17:13.807486"] [White "H1 1000"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 02:33:37.487876"] [White "H1 1000"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 02:52:15.775194"] [White "H1 1000"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 03:04:44.312286"] [White "H1 1000"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-13 03:11:03.540830"] [White "H1 1000"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-13 15:49:42.963906"] [White "H1 1000"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 16:42:20.192982"] [White "H1 1000"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-13 16:49:58.140211"] [White "H1 1000"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 15:55:58.736639"] [White "H1 1000"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 15:58:44.299319"] [White "H1 1000"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 16:02:07.821846"] [White "H1 1000"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 16:16:59.550529"] [White "H1 1000"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-14 10:44:23.917748"] [White "H1 1000"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 15:45:05.057384"] [White "H1 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 15:45:05.057384"] [White "H1 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 15:45:05.073006"] [White "H1 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 15:45:05.073006"] [White "H1 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 15:46:34.760707"] [White "H1 1000"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 01:36:53.984096"] [White "H1 1200"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 03:18:41.791167"] [White "H1 1200"] [Black "H1 1000"] [Result "1-0"] .

[Date "2019-05-13 02:37:55.616850"] [White "H1 1200"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 02:56:11.139068"] [White "H1 1200"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 03:07:47.149950"] [White "H1 1200"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-13 03:14:37.821696"] [White "H1 1200"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-13 17:01:46.893910"] [White "H1 1200"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 17:24:19.784844"] [White "H1 1200"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-13 17:28:09.587846"] [White "H1 1200"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 17:08:31.133450"] [White "H1 1200"] [Black "H2 200"] [Result

"1-0"] .

[Date "2019-05-13 17:11:14.429410"] [White "H1 1200"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 17:18:33.714111"] [White "H1 1200"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 17:21:10.588056"] [White "H1 1200"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-13 20:32:04.594733"] [White "H1 1200"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 14:42:40.651265"] [White "H1 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 14:42:40.651265"] [White "H1 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 14:42:40.651265"] [White "H1 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 14:42:40.651265"] [White "H1 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 14:42:40.651265"] [White "H1 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-13 02:05:37.641503"] [White "H1 200"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 02:31:51.412135"] [White "H1 200"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 02:36:01.576174"] [White "H1 200"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 02:13:50.637708"] [White "H1 200"] [Black "H1 400"] [Result "0-1"] .

[Date "2019-05-13 02:21:44.896859"] [White "H1 200"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 02:25:54.164797"] [White "H1 200"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 12:12:56.530194"] [White "H1 200"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 13:01:21.668023"] [White "H1 200"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 13:04:06.247964"] [White "H1 200"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 12:24:14.961188"] [White "H1 200"] [Black "H2 200"] [Result "0-1"] .

[Date "2019-05-13 12:29:14.146946"] [White "H1 200"] [Black "H2 400"] [Result "0-1"] .

[Date "2019-05-13 12:43:49.905060"] [White "H1 200"] [Black "H2 600"] [Result "0-1"] .

[Date "2019-05-13 12:49:26.232096"] [White "H1 200"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 11:09:45.249564"] [White "H1 200"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 19:24:27.332784"] [White "H1 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 19:24:48.102509"] [White "H1 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 19:24:48.102509"] [White "H1 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 19:25:10.354975"] [White "H1 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 19:25:17.712629"] [White "H1 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 01:54:45.391561"] [White "H1 400"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 02:50:40.457760"] [White "H1 400"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 02:54:21.634212"] [White "H1 400"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 02:16:21.902774"] [White "H1 400"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 02:40:32.028441"] [White "H1 400"] [Black "H1 600"] [Result

"0-1"] .

[Date "2019-05-13 02:45:31.974640"] [White "H1 400"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 13:09:55.067215"] [White "H1 400"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 13:55:15.115340"] [White "H1 400"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 13:58:37.408960"] [White "H1 400"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 13:17:16.425484"] [White "H1 400"] [Black "H2 200"] [Result "0-1"] .

[Date "2019-05-13 13:33:07.603515"] [White "H1 400"] [Black "H2 400"] [Result "0-1"] .

[Date "2019-05-13 13:42:11.015936"] [White "H1 400"] [Black "H2 600"] [Result "0-1"] .

[Date "2019-05-13 13:51:28.645154"] [White "H1 400"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 11:00:59.853297"] [White "H1 400"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 18:28:25.855082"] [White "H1 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 18:28:25.855082"] [White "H1 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 18:28:59.447215"] [White "H1 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 18:29:19.134569"] [White "H1 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 18:29:25.525917"] [White "H1 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 01:49:06.149342"] [White "H1 600"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 03:03:02.597588"] [White "H1 600"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 03:06:35.548619"] [White "H1 600"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 02:23:29.382711"] [White "H1 600"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 02:42:29.981300"] [White "H1 600"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 02:58:51.619405"] [White "H1 600"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 14:01:56.357969"] [White "H1 600"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 15:10:06.162725"] [White "H1 600"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 15:16:27.884458"] [White "H1 600"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 14:05:49.103323"] [White "H1 600"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 14:26:46.469500"] [White "H1 600"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 14:32:29.221552"] [White "H1 600"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 14:38:34.369554"] [White "H1 600"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 10:55:25.661114"] [White "H1 600"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 17:34:28.558310"] [White "H1 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 17:34:28.558310"] [White "H1 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 17:34:28.558310"] [White "H1 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 17:34:40.604243"] [White "H1 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 17:34:48.214014"] [White "H1 600"] [Black "baseline MCTS

(10000)"] [Result "0-1"] .

[Date "2019-05-13 01:45:46.255690"] [White "H1 800"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 03:09:53.813889"] [White "H1 800"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 03:13:13.493125"] [White "H1 800"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 02:27:56.283170"] [White "H1 800"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 02:47:26.097647"] [White "H1 800"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 03:00:38.215981"] [White "H1 800"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-13 15:19:10.024845"] [White "H1 800"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 15:37:27.712918"] [White "H1 800"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 15:44:43.571846"] [White "H1 800"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 15:21:56.319896"] [White "H1 800"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 15:25:15.466545"] [White "H1 800"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 15:28:25.947237"] [White "H1 800"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 15:31:14.881409"] [White "H1 800"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-14 10:50:55.653867"] [White "H1 800"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 16:38:46.260174"] [White "H1 800"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 16:38:46.260174"] [White "H1 800"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 16:38:46.260174"] [White "H1 800"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 16:38:46.260174"] [White "H1 800"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 16:39:16.806424"] [White "H1 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 11:22:49.043012"] [White "H2 0"] [Black "H1 0"] [Result "0-1"] .

[Date "2019-05-13 15:50:57.945914"] [White "H2 0"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:02:56.634249"] [White "H2 0"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 12:16:55.377829"] [White "H2 0"] [Black "H1 200"] [Result "0-1"] .

[Date "2019-05-13 13:11:45.947415"] [White "H2 0"] [Black "H1 400"] [Result "0-1"] .

[Date "2019-05-13 14:03:29.041255"] [White "H2 0"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 15:20:27.925181"] [White "H2 0"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 18:16:00.131347"] [White "H2 0"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 18:23:06.836811"] [White "H2 0"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 17:36:33.805391"] [White "H2 0"] [Black "H2 200"] [Result "0-1"] .

[Date "2019-05-13 17:57:30.025009"] [White "H2 0"] [Black "H2 400"] [Result "0-1"] .

[Date "2019-05-13 18:02:50.400062"] [White "H2 0"] [Black "H2 600"] [Result "0-1"] .

[Date "2019-05-13 18:09:31.757944"] [White "H2 0"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 03:55:07.322349"] [White "H2 0"] [Black "Hexy (expert)"]

[Result "0-1"] .

[Date "2019-05-11 12:01:43.497182"] [White "H2 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 12:02:27.255355"] [White "H2 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 12:02:27.264330"] [White "H2 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 12:02:27.270314"] [White "H2 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 12:03:57.200339"] [White "H2 0"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 12:03:16.041876"] [White "H2 1000"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 16:43:26.649989"] [White "H2 1000"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:25:29.698274"] [White "H2 1000"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 13:02:27.358103"] [White "H2 1000"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 13:56:27.429037"] [White "H2 1000"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 15:11:39.530498"] [White "H2 1000"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-13 15:39:01.619823"] [White "H2 1000"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-13 18:17:12.711501"] [White "H2 1000"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:52:50.482337"] [White "H2 1000"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 20:04:16.499386"] [White "H2 1000"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 20:20:55.376087"] [White "H2 1000"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 20:34:38.374534"] [White "H2 1000"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 20:46:41.443999"] [White "H2 1000"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-14 03:34:03.560715"] [White "H2 1000"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 21:37:11.119733"] [White "H2 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 21:37:11.119733"] [White "H2 1000"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 21:38:05.167156"] [White "H2 1000"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 21:38:28.776160"] [White "H2 1000"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 21:38:34.933132"] [White "H2 1000"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 12:06:30.271782"] [White "H2 1200"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 16:52:19.424215"] [White "H2 1200"] [Black "H1 1000"] [Result "1-0"] .

[Date "2019-05-13 17:30:08.724156"] [White "H2 1200"] [Black "H1 1200"] [Result "1-0"] .

[Date "2019-05-13 13:05:12.658036"] [White "H2 1200"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 13:59:55.343055"] [White "H2 1200"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 15:17:29.655175"] [White "H2 1200"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-13 15:45:47.774608"] [White "H2 1200"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-13 18:24:05.282108"] [White "H2 1200"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:53:56.549862"] [White "H2 1200"] [Black "H2 1000"] [Result

"1-0"] .

[Date "2019-05-13 20:07:47.940280"] [White "H2 1200"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 20:24:11.714057"] [White "H2 1200"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 20:40:32.561448"] [White "H2 1200"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-13 20:49:42.504255"] [White "H2 1200"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-13 21:21:12.539584"] [White "H2 1200"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 20:37:15.573231"] [White "H2 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 20:37:15.588859"] [White "H2 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 20:37:15.588859"] [White "H2 1200"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 20:38:27.299880"] [White "H2 1200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 20:39:17.845539"] [White "H2 1200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 11:38:08.022169"] [White "H2 200"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 15:57:17.242368"] [White "H2 200"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:09:42.379815"] [White "H2 200"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 12:26:39.417151"] [White "H2 200"] [Black "H1 200"] [Result "0-1"] .

[Date "2019-05-13 13:20:14.891179"] [White "H2 200"] [Black "H1 400"] [Result "0-1"] .

[Date "2019-05-13 14:06:58.723383"] [White "H2 200"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 15:23:05.159472"] [White "H2 200"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 17:39:06.194345"] [White "H2 200"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:02:53.799600"] [White "H2 200"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 20:06:23.946481"] [White "H2 200"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 19:37:58.649780"] [White "H2 200"] [Black "H2 400"] [Result "0-1"] .

[Date "2019-05-13 19:51:43.070046"] [White "H2 200"] [Black "H2 600"] [Result "0-1"] .

[Date "2019-05-13 19:58:56.502966"] [White "H2 200"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 03:49:48.225323"] [White "H2 200"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-11 03:10:19.915231"] [White "H2 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 03:10:19.915231"] [White "H2 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 03:10:37.714643"] [White "H2 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 03:10:50.088972"] [White "H2 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 03:11:12.306826"] [White "H2 200"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 11:43:28.096093"] [White "H2 400"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 16:00:03.034333"] [White "H2 400"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:12:19.280726"] [White "H2 400"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 12:31:36.460845"] [White "H2 400"] [Black "H1 200"] [Result

"1-0"] .

[Date "2019-05-13 13:36:26.887341"] [White "H2 400"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 14:28:05.013734"] [White "H2 400"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 15:26:25.602439"] [White "H2 400"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 18:00:13.711542"] [White "H2 400"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:19:48.910016"] [White "H2 400"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 20:23:02.826862"] [White "H2 400"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 19:40:51.415719"] [White "H2 400"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 20:11:06.952598"] [White "H2 400"] [Black "H2 600"] [Result "0-1"] .

[Date "2019-05-13 20:16:25.499412"] [White "H2 400"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 03:47:56.645525"] [White "H2 400"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-11 01:53:08.964588"] [White "H2 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 01:53:42.352498"] [White "H2 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 01:54:07.982731"] [White "H2 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 01:54:23.983133"] [White "H2 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 01:54:33.587679"] [White "H2 400"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 11:53:50.397971"] [White "H2 600"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 16:03:12.867684"] [White "H2 600"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:19:56.132408"] [White "H2 600"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 12:46:18.748371"] [White "H2 600"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 13:44:50.676984"] [White "H2 600"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 14:33:53.693622"] [White "H2 600"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 15:29:33.447871"] [White "H2 600"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-13 18:05:12.506428"] [White "H2 600"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:33:28.861372"] [White "H2 600"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 20:39:22.641265"] [White "H2 600"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 19:54:52.597051"] [White "H2 600"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 20:14:13.752723"] [White "H2 600"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 20:27:31.334969"] [White "H2 600"] [Black "H2 800"] [Result "0-1"] .

[Date "2019-05-14 03:41:40.104597"] [White "H2 600"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-11 00:22:18.846887"] [White "H2 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 00:23:45.218737"] [White "H2 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 00:23:45.225720"] [White "H2 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-11 00:24:04.370231"] [White "H2 600"] [Black "baseline MCTS

(10000)"] [Result "0-1"] .

[Date "2019-05-11 00:24:15.209869"] [White "H2 600"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-13 11:57:49.187781"] [White "H2 800"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-13 16:18:09.759736"] [White "H2 800"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-13 17:22:21.994861"] [White "H2 800"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-13 12:51:08.722830"] [White "H2 800"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-13 13:53:13.295249"] [White "H2 800"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-13 14:39:39.566583"] [White "H2 800"] [Black "H1 600"] [Result "0-1"] .

[Date "2019-05-13 15:33:46.090942"] [White "H2 800"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-13 18:10:44.714358"] [White "H2 800"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-13 20:45:01.503258"] [White "H2 800"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-13 20:48:36.688568"] [White "H2 800"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-13 20:00:50.518710"] [White "H2 800"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-13 20:17:58.554177"] [White "H2 800"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-13 20:29:13.304363"] [White "H2 800"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-14 03:39:44.619948"] [White "H2 800"] [Black "Hexy (expert)"] [Result "0-1"] .

[Date "2019-05-10 22:50:53.064157"] [White "H2 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 22:51:21.390933"] [White "H2 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 22:51:43.392037"] [White "H2 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 22:52:14.919537"] [White "H2 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-10 22:52:23.427685"] [White "H2 800"] [Black "baseline MCTS (10000)"] [Result "0-1"] .

[Date "2019-05-14 11:13:52.380183"] [White "Hexy (expert)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-14 10:41:49.833849"] [White "Hexy (expert)"] [Black "H1 1000"] [Result "1-0"] .

[Date "2019-05-13 20:50:54.049995"] [White "Hexy (expert)"] [Black "H1 1200"] [Result "1-0"] .

[Date "2019-05-14 11:05:28.249879"] [White "Hexy (expert)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-14 10:58:58.375600"] [White "Hexy (expert)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-14 10:53:44.967204"] [White "Hexy (expert)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-14 10:48:33.760192"] [White "Hexy (expert)"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-14 03:53:24.084149"] [White "Hexy (expert)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-14 03:36:26.855174"] [White "Hexy (expert)"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-13 21:06:27.171413"] [White "Hexy (expert)"] [Black "H2 1200"] [Result "1-0"] .

[Date "2019-05-14 03:51:22.188190"] [White "Hexy (expert)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-14 03:45:44.645733"] [White "Hexy (expert)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-14 03:44:06.485569"] [White "Hexy (expert)"] [Black "H2 600"]

[Result "1-0"] .

[Date "2019-05-14 03:38:06.636263"] [White "Hexy (expert)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-14 03:21:30.100800"] [White "Hexy (expert)"] [Black "baseline MCTS (10000)"] [Result "1-0"] .

[Date "2019-05-10 20:25:52.572498"] [White "baseline MCTS (10000)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-10 20:25:00.194543"] [White "baseline MCTS (10000)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-10 20:25:45.103260"] [White "baseline MCTS (10000)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-10 20:25:45.103260"] [White "baseline MCTS (10000)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-10 20:25:45.103260"] [White "baseline MCTS (10000)"] [Black "H1 0"] [Result "1-0"] .

[Date "2019-05-10 15:53:38.712920"] [White "baseline MCTS (10000)"] [Black "H1 1000"] [Result "1-0"] .

[Date "2019-05-10 15:52:10.253143"] [White "baseline MCTS (10000)"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-10 15:52:10.253143"] [White "baseline MCTS (10000)"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-10 15:52:10.253143"] [White "baseline MCTS (10000)"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-10 15:52:10.253143"] [White "baseline MCTS (10000)"] [Black "H1 1000"] [Result "0-1"] .

[Date "2019-05-10 14:48:46.912760"] [White "baseline MCTS (10000)"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-10 14:48:21.329588"] [White "baseline MCTS (10000)"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-10 14:48:21.329588"] [White "baseline MCTS (10000)"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-10 14:48:21.329588"] [White "baseline MCTS (10000)"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-10 14:48:46.912760"] [White "baseline MCTS (10000)"] [Black "H1 1200"] [Result "0-1"] .

[Date "2019-05-10 19:31:49.166070"] [White "baseline MCTS (10000)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-10 19:31:03.117375"] [White "baseline MCTS (10000)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-10 19:31:25.792915"] [White "baseline MCTS (10000)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-10 19:31:41.682054"] [White "baseline MCTS (10000)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-10 19:31:41.682054"] [White "baseline MCTS (10000)"] [Black "H1 200"] [Result "1-0"] .

[Date "2019-05-10 18:36:33.026753"] [White "baseline MCTS (10000)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-10 18:35:34.571555"] [White "baseline MCTS (10000)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-10 18:35:34.587179"] [White "baseline MCTS (10000)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-10 18:36:10.384647"] [White "baseline MCTS (10000)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-10 18:36:33.011147"] [White "baseline MCTS (10000)"] [Black "H1 400"] [Result "1-0"] .

[Date "2019-05-10 17:41:11.963739"] [White "baseline MCTS (10000)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-10 17:40:25.883406"] [White "baseline MCTS (10000)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-10 17:40:47.853666"] [White "baseline MCTS (10000)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-10 17:40:47.869270"] [White "baseline MCTS (10000)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-10 17:41:11.963739"] [White "baseline MCTS (10000)"] [Black "H1 600"] [Result "1-0"] .

[Date "2019-05-10 16:47:57.958607"] [White "baseline MCTS (10000)"] [Black

"H1 800"] [Result "1-0"] .

[Date "2019-05-10 16:44:57.916689"] [White "baseline MCTS (10000)"] [Black "H1 800"] [Result "0-1"] .

[Date "2019-05-10 16:47:14.135619"] [White "baseline MCTS (10000)"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-10 16:47:28.639137"] [White "baseline MCTS (10000)"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-10 16:47:28.639137"] [White "baseline MCTS (10000)"] [Black "H1 800"] [Result "1-0"] .

[Date "2019-05-11 12:18:32.217553"] [White "baseline MCTS (10000)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-11 12:16:31.551184"] [White "baseline MCTS (10000)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-11 12:17:35.449375"] [White "baseline MCTS (10000)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-11 12:18:14.515424"] [White "baseline MCTS (10000)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-11 12:18:14.520411"] [White "baseline MCTS (10000)"] [Black "H2 0"] [Result "1-0"] .

[Date "2019-05-10 21:47:27.150651"] [White "baseline MCTS (10000)"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-10 21:46:04.073363"] [White "baseline MCTS (10000)"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-10 21:47:04.635865"] [White "baseline MCTS (10000)"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-10 21:47:07.838226"] [White "baseline MCTS (10000)"] [Black "H2 1000"] [Result "0-1"] .

[Date "2019-05-10 21:47:14.916927"] [White "baseline MCTS (10000)"] [Black "H2 1000"] [Result "1-0"] .

[Date "2019-05-10 20:45:57.962804"] [White "baseline MCTS (10000)"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-10 20:44:48.321889"] [White "baseline MCTS (10000)"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-10 20:44:48.321889"] [White "baseline MCTS (10000)"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-10 20:44:48.321889"] [White "baseline MCTS (10000)"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-10 20:44:48.337511"] [White "baseline MCTS (10000)"] [Black "H2 1200"] [Result "0-1"] .

[Date "2019-05-11 03:20:06.166007"] [White "baseline MCTS (10000)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-11 03:18:27.433739"] [White "baseline MCTS (10000)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-11 03:19:04.658886"] [White "baseline MCTS (10000)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-11 03:19:35.500932"] [White "baseline MCTS (10000)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-11 03:19:55.376761"] [White "baseline MCTS (10000)"] [Black "H2 200"] [Result "1-0"] .

[Date "2019-05-11 02:05:04.934470"] [White "baseline MCTS (10000)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-11 02:03:25.853726"] [White "baseline MCTS (10000)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-11 02:04:05.819965"] [White "baseline MCTS (10000)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-11 02:04:34.975040"] [White "baseline MCTS (10000)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-11 02:04:53.810576"] [White "baseline MCTS (10000)"] [Black "H2 400"] [Result "1-0"] .

[Date "2019-05-11 00:38:53.216327"] [White "baseline MCTS (10000)"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-11 00:37:53.836024"] [White "baseline MCTS (10000)"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-11 00:37:53.840013"] [White "baseline MCTS (10000)"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-11 00:38:21.416679"] [White "baseline MCTS (10000)"] [Black

"H2 600"] [Result "1-0"] .

[Date "2019-05-11 00:38:41.656941"] [White "baseline MCTS (10000)"] [Black "H2 600"] [Result "1-0"] .

[Date "2019-05-10 23:01:24.962699"] [White "baseline MCTS (10000)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-10 23:00:48.379174"] [White "baseline MCTS (10000)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-10 23:00:48.385157"] [White "baseline MCTS (10000)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-10 23:00:48.394134"] [White "baseline MCTS (10000)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-10 23:00:48.399121"] [White "baseline MCTS (10000)"] [Black "H2 800"] [Result "1-0"] .

[Date "2019-05-14 03:29:46.853607"] [White "baseline MCTS (10000)"] [Black "Hexy (expert)"] [Result "0-1"] .

E. Training loss rate for H2 after 2400 training iterations

Below is a graph of the training losses for H2, after it has played 12,000 games of self-play, and done 2,400 training iterations.

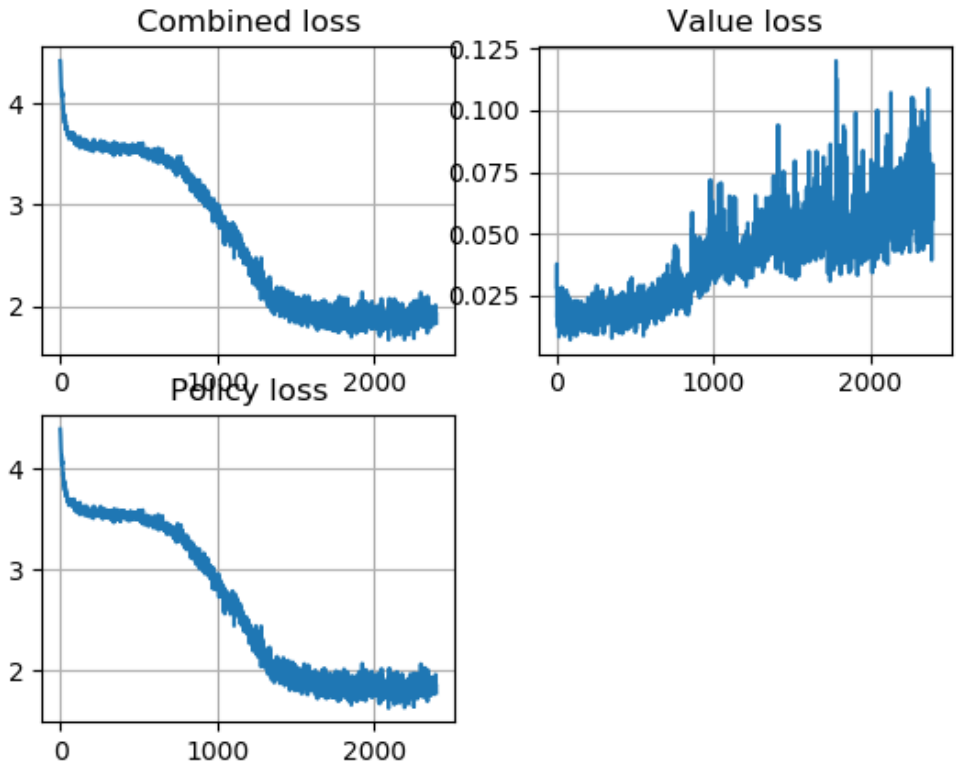

Figure E.9: Training loss rates for H2 after 2,400 training iterations.