BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

PLAYING GOMOKU WITH NEURAL NETWORKS

NEURONOVÉ SÍTĚ PRO HRU GOMOKU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR / **AUTOR PRÁCE**	**MICHAL SLÁVKA**
SUPERVISOR / **VEDOUCÍ PRÁCE**	**MICHAL HRADIŠ, Ph.D.**

BRNO 2019

Department of Computer Graphics and Multimedia (DCGM) Academic year 2018/2019

Bachelor's Thesis Specification

21764

Student:	**Slávka Michal**
Programme:	Information Technology
Title:	**Playing Gomoku with Neural Networks**
Category:	Image Processing

Assignment:

1. Familiarize yourself with convolutional neural networks and with search algorithms used for zero-sum, perfect information games.
2. Study state-of-the-art algorithms that use neural networks and state-space search to play such games.
3. Design a method for gomoku inspired by one of the state-of-the-art algorithms.
4. Create an interface for an existing gomoku tournament software.
5. Implement the proposed method and evaluate it against existing gomoku solvers.
6. Discuss the results and possible future work.
7. Create a short video presenting your work, the selected methods and your results.

Recommended literature:

- D. Silver et al.: Mastering the game of Go with Deep Neural Networks & Tree Search. Nature 2016.
- David Silver et al.: Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm, arXiv:1712.01815, 2017.

Requirements for the first semester:

- Items 1 to 4.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Hradiš Michal, Ing., Ph.D.**
Head of Department:	Černocký Jan, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 15, 2019
Approval date:	May 14, 2019

Abstract

This thesis explores the usage of AlphaZero algorithm for game of Gomoku. AlphaZero is a reinforcement learning algorithm, which does not require any existing datasets and is able to improve only by using self-play. It uses a tree search for policy improvement, which is subsequently used for training. This approach was able to defeat the previous state of the art methods. Generating training data of high quality requires a lot of computationally expensive iterations, which makes them algorithm slow to train. Experiments show that the strength of the play is growing with each subsequent iteration, this might indicate that it still has room for improvement with more training and that it has not reached its full potential.

Abstrakt

Táto práca sa zaoberá použitím algoritmu AlphaZero pre hru Gomoku. AlphaZero je založený na spätnoväzbnom učení a k trénovaniu nemusia byť využité žiadne existujúce datasety. Trénovanie prebieha iba na hrách algoritmu samého so sebou. AlphaZero používa algoritmus na prehľadávanie stromu, pre zlepšenie stratégie. Na vylepšnej stratégii sa následne trénuje neurónová sieť. Tento prístup bol úspešný v hrách proti existujúcim algoritmom. Generovanie trénovacích dát vysokej kvality si vyžaduje veľa výpočetne náročných iterácií trénovania a generovania dát. Experimenty ukázali, že každou iteráciou sa algoritmus zlepšuje, čo naznačuje, že je ešte miesto na zlepšenie, ale množstvo iterácií nedostačovalo na to, aby bol poriadne natrénovaný.

Keywords

neural networks, Monte Carlo tree search, AlphaZero, backpropagation, reinforcement learning

Kľúčové slová

neurónové siete, Monte Carlo tree search, AlphaZero, backpropagation, reinforcement learning

Reference

SLÁVKA, Michal. *Playing Gomoku with Neural Networks.* Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Michal Hradiš, Ph.D.

Rozšírený abstrakt

Táto práca sa zaoberá vytvorením algoritmu, ktorý by bol schopný hrať hru Gomoku. Pre tento účel existujú rôzne prístupy, ja používam algoritmus AlphaZero, ktorý dosiahol veľmi dobré výsledky v šachu, hre Go a japonskom šachu shogi. Vo všetkých týchto hrách jednoznačne prekonal existujúce algoritmy. AlphaZero nepoužíva žiadne ručne vytvorené heuristiky, ale požíva algoritmus na prehľadávanie stromu, ktorý kombinuje s neurónovou sieťou zefektívňujúcou prehľadávanie. Avšak tento algoritmus je revolučný v tom, že nevyužíva žiadne existujúce dáta pre natrénovanie neurónovej siete. Namiesto toho využíva spätnoväzbové učenie, pričom sa spolieha na zlepšenie stratégie stromovým prehľadávaním. Algoritmus požitý na stromové prehľadávanie sa volá Monte Carlo tree search.

Monte Carlo tree search algoritmus vykonáva simulácie, pri ktorých si vytvára herný strom. Pri vykonávaní simulácií vyberá ťahy podľa toho, ako skončili predošlé simulácie, pričom vyberá tie, ktoré pri prehľadávaní viedli k výhre a zároveň sa snaží vybalancovať výber najperspektívnejších ťahov s prieskumom iných ťahov. Neurónová sieť poskytuje odporúčania ťahov, na ktoré sa má prehľadávanie sústrediť, a ohodnotenia hracích plôch, čím drasticky znižuje množstvo simulácií, potrebných na nájdenie optimálnej stratégie.

Trénovanie sa deje v dvoch krokoch: generovanie dát a optimalizácia parametrov na vygenerovaných dátach. Dáta sa generujú tým, že algoritmus hrá sám proti sebe. Vygenerované dáta obsahujú reprezentáciu stavu hracej plochy, stratégiu vylepšenú stromovým prehľadávaním a výsledok odohranej hry. Neurónová sieť sa učí túto stratégiu a predpovedať výsledok hry na základe stavu hracej plochy. Pre natrénovanie siete je potrebné vykonať veľké množstvo trénovacích krokov, generácií, čo je pomerne výpočetne náročné.

Na implementáciu neurónovej siete som použil knižnicu na hlboké učenie Pytorch, ktorá je primarne určená pre jazyk Python, ale obsahuje aj c++ rozhranie. Kvôli efektivite implementácie Monte Carlo tree search algoritmu som sa rozhodol použiť jazyk c++, pretože implementácia v Pythone bola príliš pomalá. Týmto som dosiahol až 30-násobné zrýchlenie. Pre evaluáciu neurónovej siete som využil c++ rozhranie Pytorch knižnice. Aby som nestratil možnosť jednoducho testovať rôzne konfigurácie, vytvoril som z mojej Monte Carlo tree search implementácie Python rozšírenie.

Algoritmus som trénoval na 30 generáciach dát a celkovo pri trénovaní bolo vygenerovanych viac ako 1 gigabyte dát, čo zodpovedá viac ako pol miliónu hracích pozícií. Keďže pravidlá Gomoku nerozlišujú rotácie hracej plochy, každá pozícia ma osem rotácií.

Z experimentov som zistil, že počet simulácií naozaj zlepšuje stratégiu a s počtom iterácií rastie sila algoritmu. Pri hre algoritmu samého so sebou, pričom jedna instancia používala 1000 simulácií pre jeden ťah a druhá používala 2000 a 5000 iterácií, v prvom prípade instancia s väčším počtom simulácií vyhrala 61% hier a v druhom 71% hier.

Moj algoritmus som taktiež porovnával proti iným existujúcim programom, ale zatiaľ ich nebol schopný poraziť. Taktiež pri hre voči ľudským hračom nedosiahol veľmi dobré výsledky, avšak pri porovnávaní parametrov z rôznych generácií, novšie generácie boli progresívne silnejšie. Konštantný rast v sile novších generácii naznačuje, že trénovanie ešte nedosiahlo svoj limit a pri ďalšom trénovaní by sa algoritmus ešte mohol zlepšiť.

Playing Gomoku with Neural Networks

Declaration

I declare that this Bachelor's thesis is my original work and that I have written it under the guidance of Michal Hradiš, Ph.D. All sources and literature that I have used during my work on the thesis are correctly cited with complete reference to the respective sources.

. . . .
Michal Slávka
May 15, 2019

Acknowledgements

I would like to thank Michal Hradiš for his guidance and advises.
Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Contents

Chapter 1

Introduction

This thesis explores a combination of neural network and reinforcement learning for perfect information game and how this approach can achieve superior results in previously known methods. I am using the game of *Gomoku* to show how these two algorithms combine into the *AlphaZero* algorithm introduced by David Silver et al. [27] which is the successor of algorithms for playing game *Go*, *AlphaGo Lee* and *AlphaGo Zero* [29] algorithms which were the first computer programs ever to beat top human players in this game. I have the chosen game *Gomoku* because of its relative simplicity and which should correspond with comparatively lower training times.

One of the biggest breakthroughs in computer programs playing against human players was when DeepBlue [5] defeated the human champion in *Chess*. This program used the Alpha-Beta pruning algorithm [14] in a combination with handcrafted features, heuristics, and database of games. This approach relied heavily on brute-force computation and domain-specific knowledge, which made it hard to adapt existing programs to play different games. Games with bigger search space, like *Go*, were considered intractable problems and computers were able to play at most at an amateur level. *Chess* has game-tree complexity [36] about 10^{120} compared to *Go* which has game complexity of 10^{360}.

In the following chapters, I will describe existing methods for solving games, their principles, and differences. A detailed description of the Monte Carlo tree search and the *AlphaZero* algorithm is followed by a description of my version of this algorithm and my implementation. In the end, there are experiments with the performance of the algorithm.

Chapter 2

Playing games

In my work, I am interested in two-player, zero-sum, perfect information games. In the following text, the word game is referring to this type of game. A zero-sum[37] game is a game in which if all losses were subtracted from all gains for each player, it would sum to zero, in other words, a player is winning by the same margin as the other player is loosing. A perfect information[35] games are such games, in which every player has all the information about the state of the game. Examples of this kind of games are *Gomoku*, *Chess*, *Go* or *Shogi*.

Creating game-playing computer programs was always an interesting subject in computer science. A special place is held by board games, that humans were trying to master for centuries. With improvements in artificial intelligence computers were able to play on a superhuman level in games like *Chess*, although *Chess* is not a solved game and remains an open problem. This creates space for improvements because we may never solve it completely, a heuristic approach is required. Until very recently existing methods were not able to tackle games with large branching factor and without a straight forward strategy. I explore new approaches to solving games on game *Gomoku*, which is rather simple compared to the other board games.

2.1 Gomoku rules

Gomoku is a two-player game with simple rules very similar to tic-tac-toe. Both players have absolute knowledge about the state of the game, perfect information. The game is played on board with vertical and diagonal lines. Each player then places stones on intersections of the lines. Players can only play moves that were not previously played. In the official rules winner is the first player who succeeds in placing exactly five stones in a row, column or diagonally next to each other. If there is a sequence of more then five stones of the same color in a row, the game continues normally but it does not count as a winning sequence. Another variant of the game is called free-style *Gomoku* where the legal winning sequence is at least five or more stones in a row. Board usually has dimensions 15×15 but also board with dimensions 19×19 can be used.

For my experiments, I used freestyle rules with board size 13×13 to reduce complexity even more.

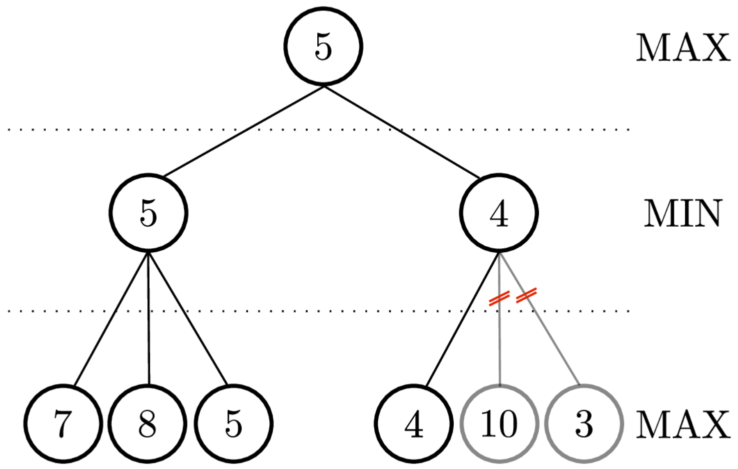

Figure 2.1: Example of *Alpha-Beta pruning*. Grey nodes would be explored in *Mini-Max* but the value of root node would not change.

2.2 Existing approaches

To create a computer program for playing the games with reasonable strength, numerous methods exist. The simplest method and probably the most intuitive for zero-sum games is called *Mini-Max*. This algorithm tries to find a move, which will lead to maximal reward in a case next player chooses the best possible move. Therefore making a move leading to minimal guaranteed reward. To find such a move algorithm is searching a game tree up to specified depth or until a terminal state is encountered or depth limit is reached. When the depth limit is reached the exact value of the state is not know and is approximated using a heuristic. This algorithm without a depth limit will provide optimal move, however, this approach can be computationally infeasible because of its time complexity which grows exponentially $O(b^d)$, where b is a branching factor of the game and d is maximal depth of search.

To make *Mini-Max* more efficient some states can be skipped during the search. An algorithm that is trying to reduce the number of visited nodes that otherwise would be visited with *Mini-Max* search is called *Alpha-Beta pruning* [7]. This algorithm was rediscovered a couple of times and perfected by Knuth and Moore [14]. The idea behind it is to stop searching a branch in a game tree if there is at least one move possibly worse than the previously found move. This is done by maintaining two values, *alpha*, and *beta*, which are initially set to negative infinity and positive infinity respectively. *Alpha* keeps track of the minimal score of the maximizing player and *beta* keeps track of the maximal score of the minimizing player. The search is stopped if $alpha \geqslant beta$.Comparison of search tree build by *Alpha-Beta pruning* and *Mini-Max* is in the figure 2.1. In the best-case scenario, this can reduce the time complexity to $O(b^{\frac{d}{2}})$ in the worst-case it is the same as in *Mini-Max*.

In many games, *Alpha-Beta* pruning based algorithms achieved very high ranking. For example open sources *Chess* engine *Stockfish* [32] or *Shogi* engine *elmo* are one of the strongest engines [26, 31] and utilize *Alpha-Beta pruning* along with game databases and sophisticated handcrafted heuristics. Other options based on *Mini-Max* algorithm is for example $MDT(f)$ algorithm [22] which runs multiple instances of Alpha-Beta search.

Although *Alpha-Beta pruning* algorithms were able to surpass human players in *Chess*, in games with large search space this approach was not good enough to play at a superhuman level. In the game of game *Go* computer programs were able to play on an amateur level and defeated professional players in a couple of games on board size 9×9 [34].

With the advancement in machine learning, using neural networks trained to match policies from existing data sets or to estimate the value of a state, achieved some very good results.

Algorithm *DeepChess* [6] uses two dis-joined pre-trained auto-encoders to create a vector representation of the position in *Chess*, followed by another fully connected layers. It was trained with supervised learning to compare two positions and predict which one is more likely to win. This algorithm achieved a strong grandmaster level.

In game *Go* first major breakthrough was done by David Silver at al. with algorithm *AlphaGo* [27], the first algorithm able to beat a professional human player in a standard game with 19×19 board. This algorithm used a combination of Monte Carlo tree search [15] and two separate convolutional neural networks for policy and evaluation of game state, the policy network was trained on a database of games played by human masters and value network was then trained by reinforcement learning by playing games against itself with already trained policy network. *AlphaGo* was able to defeat world champion Lee Sedol in five-game match losing just one game, which was compared to famous Garry Kasparov versus IBM's Deep Blue match held in 1997. This algorithm was later succeeded by even stronger *AlphaGo Zero* [29]. *AlphaGo Zero* was trained by *tabula rasa* reinforcement learning algorithm, using only self-play data, without any pre-training on human games. The two networks used in *AlphaGo* were replaced by a single network with two heads, providing policy and a state value.

After achieving these successes in *Go* more general version *AlphaZero* [28] was created by the same authors. This is a more general version of *AlphaGo Zero*, trained entirely by reinforcement learning, without using any existing dataset. *AlphaZero* was created to master *Chess* and *Shogi* (also known as Japanese chess) but also achieved better results in *Go*. *AlphaZero* was able to convincingly beat some of the strongest existing engines, including previously mentioned *Stockfish* in *Chess* and *elmo* in shogi.

2.3 Reinforcement Learning

Reinforcement learning (RL) is a field in machine learning dealing with how a software agent should learn to take actions in an environment with regard to a long term reward. It differs from many other machine learning areas in not having information about how actions should be taken, prior to learning. By interacting with the environment agent discovers new states and potentially gains rewards. This interaction happens discreetly in a loop, where the environment supplies the agent with representation of its state, to which the agent reacts by making an action. The environment then as a reaction to the agent's action changes its state to a new one and presents the new state to the agent with a reward if any is associated with the action state transition. This creates an action feedback loop see figure 2.2. The agent then tries to improve its policy for choosing an action in any given state by utilizing feedback from the environment and discover optimal policy. In many problems, there is no immediate reward making finding the solution much harder because the eventual reward is a result of a series of actions.

There are four main components in RL[30]:

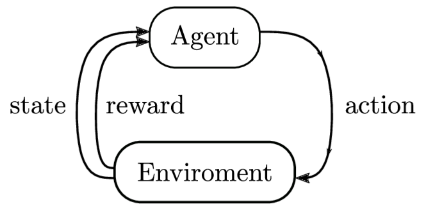

Figure 2.2: Reinforcement learning. Interaction of an agent with an environment.

- **Policy** $\pi(s)$ is a mapping from state to actions to be taken. The agent chooses action according to its policy. The goal of reinforcement learning is to find an optimal policy π_*, which leads to higher or equal long term rewards than any other policy π.

- **Reward** $r(s, a)$ is an immediate response of the environment to the agent's actions. It defines a goal which should be maximized by the agent in the long term.

- **Value function** $v(s)$ or $q(s, a)$ is a function estimating how much immediate reward from the environment can be accumulated in future based on state or state-action pair.

- **Model** of the environment allows to make assumptions about how the environment will react to actions. Models are used for planning by considering possible future situations. Models of the environment are not always available and are not conditional for RL algorithms. Models can be created be created by algorithms or can be provided for them.

Exploitation-exploration trade-off. This way of learning comes with another challenge. The agent has to balance the *exploration-exploitation tradeoff*. It has to exploit what he already learned to obtain a reward but also keep exploring to find potentially more profitable actions, from which it could improve its future strategy. If the algorithm is concentrating too much on either of those it will slow down the learning process of finding the best policy or fail completely. In case of environments reaction to actions has some randomness to it, agent can't be discouraged by not gaining a reward from taking action one time and can't be drawn to early hit too much. Each action has to be tried a number of times to get a good idea of how good it actually is.

Markov Decision Process. The environment is usually defined as a Markov Decision Process (MDP, an example in figure 2.3). MDP is an extension of the Markov chain to which it adds actions made by agent and rewards for actions. Where by choosing action in our case reward is gained as a win or loss. If MDP's action space is finite it is called a finite MDP.

Finite MDP is a 4-tuple $(S, A, p(s'|s, a), r(s, a, s'))$, where S is a finite set of states A is a fine set of actions, $p(s'|s, a)$ is a probability of transitioning to state s' from the state s given action a and $r(s, a, s')$ is reward associated with the transition from s to s' by action a.

Optimal policy. The optimal policy π^* is a policy with the expected return is better or equal to all other policies. In some cases we can easily find such policy by discovering

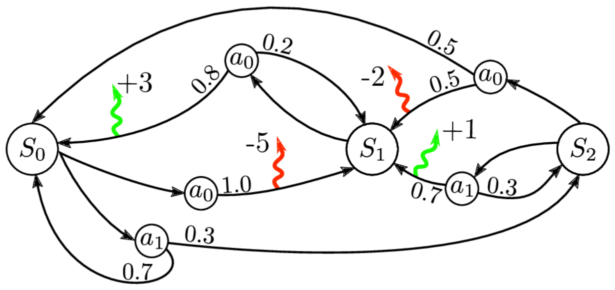

Figure 2.3: Example of Markov Decision Process

optimal value function $v^*(s)$ or $q^*(s,a)$ for all states s and all actions a, which satisfies bellman optimality equation:

$$v^*(s) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v^*(s')] \tag{1}$$

or

$$q^*(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma \max a' q^*(s',a')]. \tag{2}$$

In *AlphaZero* neural network is trying to learn *Value function* and *Policy*, which it learns directly from policies created by a tree search algorithm. The tree search improves policies from neural network and during training process neural network learns policies which are closer to optimal policy π^*.

2.4 Artificial Neuron and Neural Networks

Principle of an artificial neuron and neural networks were invented by Anthony [2] and was loosely inspired by the structure of neurons in the brain, with no intention to accurately model real brain cell. An artificial neuron is a mathematical function with arbitrary number of inputs and one output. Neurons can be connected to each other taking the output as input in the subsequent layer. A biological neuron consists of dendrites, cell, called soma, and axons. Dendrites and axons are analogous to inputs and output, respectively, of an artificial neuron. Dendrites receive signals, which creates a potential that travels through the cell and leads to excitation on axon proportional to the strength of received signals. Dendrites and axons are connected to other cells by synapses which allow to pass signal between them. Synapses may increase or decrease the strength of the signal so it selectively passes through cells.

Neuron. In the artificial neuron, the role of synapses is taken by weights. The number of weights depends on the number of inputs. Each input is multiplied by the corresponding weight. The results of multiplication are summed, bias is added and passed to activation function (see figure 2.4a). Let $g(\vec{w}, \vec{x})$ be a function of a neuron, where $\vec{x}$ is vector inputs of the function and $\vec{w}$ are weights. Usually, there is also a bias is typically denoted as w_0, so for K inputs we have $K + 1$ weights

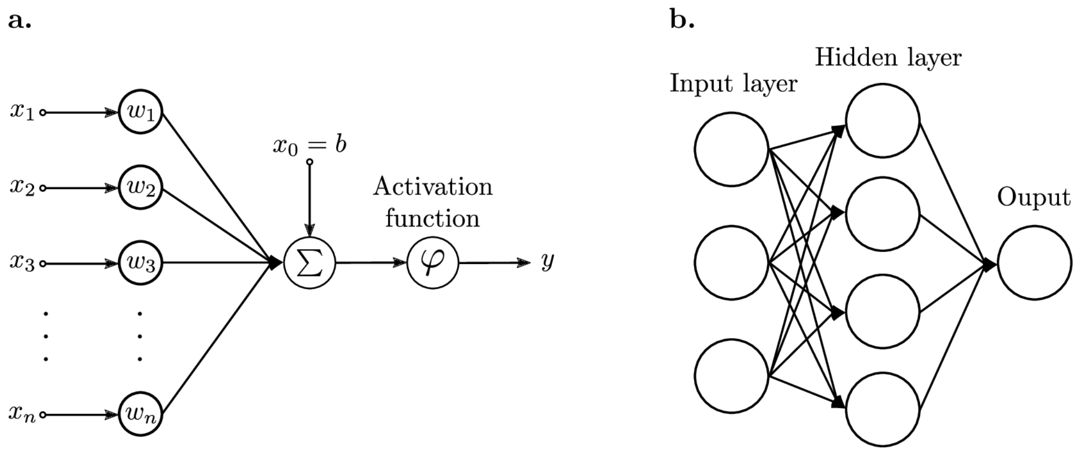

Figure 2.4: **a** Artificial neuron with input vector $\vec{x}$, weights $\vec{w}$, with activation function φ and output y. **b** Neurons connected to neural network.

$$g(\vec{w}, \vec{x}) = f(w_0 + \sum_{i=1}^{K} x_i w_i) \tag{3}$$

where f is an activation function. Example of a common activation function is logistic sigmoid denoted as φ. To be able to use the training algorithm described in 2.4 the activation function is reacquired to be differentiable.

Neural Network. When we use the output of one neuron as input for another we call it a neural network. Modern neural networks can have millions of parameters and can solve very hard problems. Commonly neurons are organized in layers, where every layer is labeled $0, 1, ..., l$ and consists of a number of neurons. In fully connected linear layers output of every neuron in layer l is input to every neuron in layer $l+1$. The first layer of the neural network is referred to as *Input layer*, there is usually one neuron per input feature. Subsequent layers are called *Hidden layers*, and the last layer is called *Output layer*. An example of a neural network with fully connected linear layers is shown in figure 2.4b. To be able to connect neurons we must impose another constraint on activation function and that is that it has to be non-linear. If used linear activation function or none at all the network would be able to learn only as much as one layer because multiplying input vector by a matrix is equal to linear transformation in space and a series of such transformations can be expressed by one linear transformation.

Learning. Adjusting weights in an artificial neuron to perform a certain task is referred to as learning. It is an optimization process where we minimize the difference between the expected outcome and output of the neuron. This difference is called cost. The cost can be computed in many ways, for example, we can use Mean Square Error (MSE)

$$C(t, y) = \frac{1}{2}(y - t)^2, \tag{4}$$

where C is MSE cost function t is expected outcome for training example $\vec{x}$ and y is the output of the neuron.

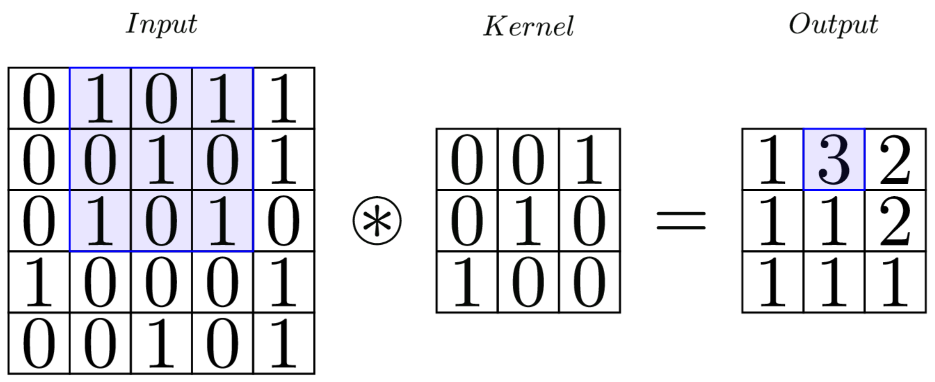

Figure 2.5: Computation of convolution with no padding, kernel size $3x3$ and stride 1. The highlighted output corresponds to highlighted input after applying kernel.

There are multiple optimization methods for finding how the weights should be adjusted. These methods are based on finding gradients of weights with respect to cost function and then adjusting the weights to decrease cost. These adjustments for each weight is done according to its gradient, also known as gradient descent. For single neuron from equation 3 using cost function from 4 computation of gradient for weight w_j using delta rule is simple

$$\frac{\partial C}{\partial w_j} = \frac{\partial C}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_j} = -(t-y)f'(z)x_j, \tag{5}$$

To find gradients when dealing with multiple layers Rumelhart in 1986 proposed backward propagation of error (backpropagation) algorithm [24] which is a generalization of delta rule. We compute gradients from the last layer to first, hence the name backpropagation. When dealing with an output layer L the computation is the same as in equation 5 but with added indexes for each output neuron. Gradient for weights in previous layer $L-1$ is

$$\frac{\partial C}{\partial w_{jk}^{L-1}} = \frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial w_{jk}^{L-1}}, \tag{6}$$

where a^L is equal to y but was used to generalize for all layers, k is an index of neuron in a layer, a^{L-1} is and output of layer $L-1$ and z^{L-1} is $\sum_{j=1}^{J} x_j k w_j k$.

Weights are then updated according to the gradient:

$$\nabla w_{jk}^{l} = \eta \frac{\partial C}{\partial w_{jk}^{l}}, \tag{7}$$

where η is the size of a step of update in the direction of gradient called learning rate.

2.5 Convolutional neural networks

Convolutional neural networks (CNN) are a type of neural network which is very well suited for image processing. Using backpropagation to train convolutional neural networks was pioneered by LeCun [19, 18]. CNN takes a little bit different approach than a standard fully connected linear network. To create a functioning image classifier there are two main properties of images that cause problems for fully connected linear networks. Two pixels next to each other are more related then pixels on a different side of the image. The fully connected linear network would have to learn this fact, in CNN this information is implicit and the receptive unit of CNN covers some area in the image. This means that CNNs are better in feature recognition in images and can have fewer parameters while performing the same as a fully connected linear network would. The second property of images that makes CNN perform better on images then fully connected linear layers is that shift in position does not change what image depicts. While CNN is position invariant, the linear network would have to learn each variation in position. CNN solves this by using multiple pixels in a surrounding area at once and performing the same operation for every group of pixels.

Convolutional neural networks were successfully used in many different areas like image recognition, image analyses, natural language processing or for playing games. Properties of the game board are very similar to the ones in the image. Game patterns can be invariant of their position and pieces that are closer together are influencing each other more than pieces further apart.

This work by using convolutional kernels, which are essentially filters are sensitive to some patterns and convoluting them over whole input. This creates an abstract representation of the input, which is then passed to the next layer. Input is a tensor representing image or board and has dimensions width, height, and a number of channels. Kernels are also matrices of fixed size, an example of how the output is computed is in figure 2.5. The goal is to adjust parameters of kernels to pick up patterns that are relevant for our task. CNNs are organized in the layer where the role of a neuron is taken by the kernel, but kernel apart from the neuron is not fixed for one input feature but is convoluted over whole input. These parameters are adjusted in the same manner as in standard neural network, this can be done because convolution is differentiable operation. The result of convolutions is then processed by the next layer. Dimensions of output are controlled by stride and padding.

Chapter 3

AlphaZero

AlphaZero algorithm [29] uses a neural network to provide for a tree search algorithm recommendations for plausible moves called *policy* and estimations of how good positions are called *value*. The search algorithm, using the neural network is then able to provide much stronger policies than neural network by itself. The tree search algorithm used for this purpose is called Monte Carlo tree search [15] (MCST).

3.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic tree search algorithm used to find optimal action for problems where a generative Markovian Decision Process exists. MCTS aim is to find a policy for an agent that will lead to the highest reward. Response to this action is provided by an environment that will present a new state to the agent and potentially provide a reward for the agent. MCTS is capable of providing near optimal actions if a generative Markov Decision Process exists for the environment.

MCTS was proposed by Abramson [1]. This algorithm became a standard method for solving two-player games. I will explain a variant of MCTS called UCB1 for Trees (UCT) introduced by Kocizs and Szepesvary [15]. General rollout based MCTS game tree is created by running simulations from the current state of the game, then choosing actions by the highest observed long term reward and traversing the existing game-tree until the leaf state is reached. The leaf state is then evaluated and reward is acquired. The reward is then accumulated for each state-action pair encountered during simulations. This allows to bias choosing better action if the state is reencountered again and potentially to converge faster to the best solution. In figure 1 is a general scheme of rollout-based MCST for a two-player game, where the reward is either 1 for a win, -1 for loss and 0 for a draw.

Each successive action is made by a different player than the previous one. This means that if during simulation we select terminal state s_T, reached from state s_{T-1} by action a_{T-1}, played by player P_1, and reward in s_T for player P_2 is r_T, the value q associated with state s_{T-1} will be updated to $q(s_{T-1}, a_{T-1}) = q(s_{T-1}, a_{T-1}) + (-r_T)$. This is applied for all previous states $q(s_{T-2}, a_{T-2}) = q(s_{T-2}, a_{T-2}) + (-(-r_T))$.

A leaf node is a node that has not been visited before. After reaching the leaf node, searching is discontinued and sample reward is generated by randomly choosing moves done by the function *Evaluate* on the line 13, until a terminal state is reached. If the terminal node is reached the reward is the result of the game. Updating of state-action pair is done on the line 21.

Algorithm 1: Monte Carlo Tree Search algorithm [15]

```
   INPUT: state, terminal condition
   OUTPUT: policy $\pi$

   /* Start                                                  */
1  Function MonteCarloTreeSearch(s, simulations):
2      for $i \leftarrow 0$ to simulations do
3          Search(s)
4      end
5      return getPolicy(s)
6
7  Function Search(s):
8      if Terminal(s) then
9          $v$ = Reward (s)
10         return $-v$
11     end
12     if Leaf(s) then
13         return $-$Evaluate(s)
14     end
15
16     a := selectAction(s, Q, N)
17     newstate := newState(s, a)
18
19     $v$ := Search(newstate)
20
       /* Updating node                                      */
21     $Q(s,a) = \frac{Q(s,a)*N(s,a)+v}{N(s,a)+1}$
22     $N(s,a) = N + 1$
23
24     return $-v$
```

UCB1 and UCT By selectively selecting actions, which look more promising we are able to narrow down search space and converge to optimum faster. To do this and leverage also possibility to miss optimal solution Kocsis and Szepsvári proposed an application of a bandit algorithm UCB1 [3]. They called this algorithm UCB1 for trees (UCT) but generally referred to as MCTS.

Bandit problems are problems where there are multiple actions. Each action will provide a random reward from a probability distribution. Bandit algorithms are trying to discover the highest grossing distribution with minimizing regret from taking sub-optimal actions, also known as exploration-exploitation trade-off. In the MCTS algorithm we want to explore actions most likely leading to winning without wasting resources on other actions (line 16). UCB1 succeeds in dealing with the exploration-exploitation problem. From all valid actions in state s it chooses an action that maximizes upper confidence bound using mean of obtained rewards for each action $Q(s, a)$ and bias which is increasing with a number of visits $U(s, a)$:

$$A_t = \operatorname*{argmax}_a \left\{ Q(s,a) + U(s,a) \right\}. \tag{8}$$

Bias is computed as:

$$U(s,a) = C_p \sqrt{\frac{2ln(\sum_b N(s,b))}{N(s,a)}}, \tag{9}$$

where $N(s,a)$ is a number of visits of edge (s,a).

As number of visits $N(s,a)$ grows probability of choosing action a will decrease. This property encourages exploration with the exploitation of early random rewards. How much will this bias decrease the significance of gained rewards is controlled with a constant C_p.

3.2 Neural network

AlphaGo Zero and *AlphaZero* algorithms residual convolutional networks [9] were used as a common core for policy head and value head. This network consisted of 20 to 40 residual blocks with batch normalization [10] and rectifier non-linearities (ReLU) [8]. The output of this network was then used in separate heads each consisting of fully connected linear layers.

Board representations. Each game requires a different board representation. Simplest rules and board representation has a game of *Go*. For the game of *Go* board is represented as planes for each player each of size of the board. Placement of game pieces is indicated by 1 on the corresponding position, zeros are elsewhere. Each input contains eight step history and one plane indicating player, which is to move.

3.3 Monte Carlo tree search and neural network

The advantages of the Monte Carlo tree search algorithm are that it is domain independent and is quite accurate on predicting near optimal actions given that enough simulations are performed. This algorithm also scored some successes in imperfect information games such as backgammon or poker. To increase performance and accuracy of MCTS it is combined with neural network f with parameters θ is used to provide both value vector of policies containing probabilities of choosing available action in given state s and scalar value v which is an estimated probability of winning a game in state s. These probabilities and value then guide the search algorithm to concentrate on more promising moves. The effect that it has on *policy* improvement can be seen in figure 3.2.

To combine general MCTS with the neural network a few adjustments need to be done. For each visited node corresponding to state s following triple is stored:

$$\{N(s,a), W(s,a), P(s,a)\}, \tag{10}$$

where $N(s,a)$ is a number of visits of edge (s,a), $W(s,a)$ is a accumulated state value and $P(s,a)$ is state-action value. The average reward than can be computed as $Q(s,a) = W(s,a)/N(s,a)$. In game *Gomoku* there is no immediate reward so in the scheme of general MCTS it is always zero and can be left out.

Again multiple simulations are performed after which action is played. Each simulation has three phases: Selection, Expansion, Evaluation, and Update.

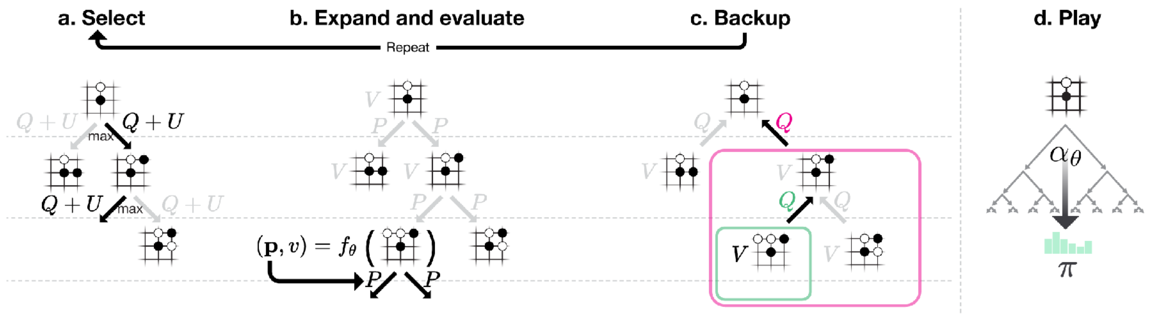

Figure 3.1: Monte Carlo tree search in combination with a neural network. Source Mastering the Game of Go without Human Knowledge[27]

Selection. When selecting actions variant of PUCT algorithm [23] is used instead of UCB1 algorithm. The PUCT algorithm as UCB1 balances exploitation and exploration but converges faster to correct solution. Actions are selected according to equation 8, but bias $U(s,a)$ is

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}, \tag{11}$$

where c_{puct} is a constant controlling level of exploration. You can see an illustration of this step in figure 3.1a.

Expansion and evaluation. Neural network provides value and policy vector $(v, \vec{p}) = f_\theta(s)$ and is evaluated and new node is added to search tree when leaf node is reached (line 13). Value v plays the same role as random play in general MCTS. However this is superior to playing randomly because the neural network is trained to estimate the probability of winning the game, this number of simulations getting a good estimate can be reduced. Policy vector serves as a recommendation for MCTS for concentrating on actions that are more likely to lead to win. This recommendation are stored for each edge $P(s,a) = (1-\epsilon)p_a + \epsilon\eta_a$, where $\eta = dirichlet(\alpha)$ is Dirichlet noise added to prior probabilities to achieve additional exploration and $\epsilon = 0.25$. Parameter α in AZ was used proportional to number of possible moves $0.3, 0.15, 0.03$ for *Chess*, *Shogi* and *Go* respectively. Figure 3.1b.

Update. In figure 1 line 24. Statistics in sequence of nodes are updated. value v provided by neural network in previous step is added to $W(s,a) = W(s,a) + v$, sign of value is alternating for each subsequent node.. Visit count is incremented by one for each node in sequence $N(s,a) = N(s,a) + 1$. Figure 3.1c.

Move selection. After finishing simulations, move is chosen according to policy $\pi(a|s_0) = N(s_0,a)^{1/\tau} / \sum_b N(s_0,b)^{1/\tau}$, which is proportional to visit counts of actions a from root state s_0. Actions are chosen randomly with associated probabilities $\pi(a|s_0)$. Temperature τ is set to 1 for number of first moves. This does not change the distribution and ensures greater variability of starting positions. For the rest of the game temperature is set to infinitesimal number $\tau \to 0$ to deterministically select the best action. Figure 3.1d.

3.4 Training

Training of neural network is done from scratch entirely by RL, starting with randomly initialized parameters. This gives the algorithm an opportunity to come up with its own strategies. We can divide training process into three steps: *Self-play*, *Optimization*, *Evaluation*. These three steps are repeated until the network is fully trained. In *AlphaZero* the evaluation step was left out to speed up the training process.

Self-play. In this phase training data is being generated by letting the algorithm play against itself with the best parameters. MCTS takes policy $\vec{p}$ and value v predictions of neural network and by performing simulation it provides improved policy π. This improved policies along with actual results of the games from self-play are then stored as triple $(s, \vec{(\pi)}, r)$, where s is a representation of the board and r is the result of the game.

Optimization. Optimization of parameters of the neural network is done on data generated during self-play, using multiple previous generations data generated by different models to avoid overfitting to one model. For *policy* optimization *cross entropy* loss was used and for *value* was used *mean square error*. Both losses were weighted equally.

Evaluation. Updated parameters $\theta + 1$ are then evaluated against parameters θ by using them in MCTS and playing against each other. If a new generation is better then previous it is used for the next round of self-play. If parameters $\theta + 1$ were not better then the previous version, more data is generated using parameters θ and used to train a new generation. During evaluation temperature τ is infinitesimal from the beginning of the game to mitigate randomness in evaluation.

Training resources Data generation was done using $5,000$ Tensor Processing Units [13] (TPU), starting with randomly generated parameters. New generations were trained for 64 seconds and in total $700,000$ steps with mini-batch size $4,096$ were performed during the training process. This configuration outperformed *Stockfish* after just 4 hours.

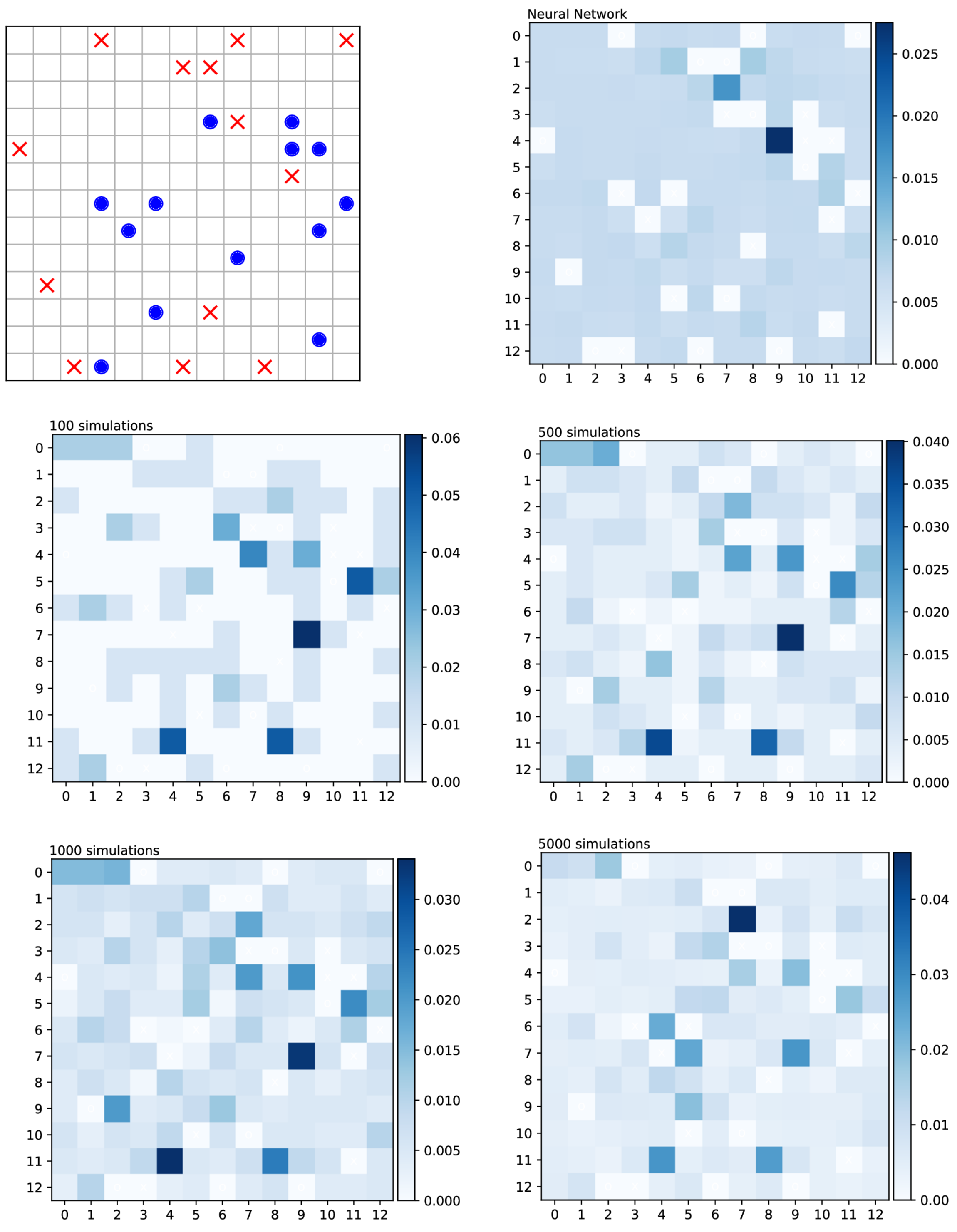

Figure 3.2: Board position, the raw output of the neural network, improved policies from Monte Carlo tree search after 100, 500, 1000, 5000 simulations respectively.

Chapter 4

AlphaZero for Gomoku

The *AlphaZero* algorithm is easy to adjust to different games. Here I will describe the choices I made in my implementation and selection of hyperparameters.

4.1 Neural network architecture

I could not match the resources used for training and generating data authors have in-disposition 3.4, so I used much smaller and simpler architecture. My network consisted exclusively of convolutional layers in the core network as in both heads. I used layers with kernel size 3, 16 output channels and padding size 1 so the width and height of output stay the same. Each layer was followed by batch normalization and ReLu. In the core network, I was using 4 layers and for each head 2 additional separate layers. The output of policy head was a probability distribution, to get that I used soft-max. Value head had another convolutional layer with 1 output channel and kernel size 1 to reduce dimensions and then average pooling layer followed by to $tanh$ non-linearity to scale output to a range $[-1, 1]$. You can see an illustration of architecture in figure 4.1.

Board representation. *Gomoku* board was represented by two planes, one for each player. Planes were in dimensions of the game board having ones indicating the presence of players stone on a given position and zero otherwise. The first plane always represents the player on move, so there is no need to indicate which player is moving. In original article history was also included but in my work, I deemed this as unnecessary.

4.2 Monte Carlo tree search configuration

Dirichlet noise. To have constant α adequate to all board sizes I created following equation to deduce proper value:

$$\alpha = \frac{actionSpace * avgLen - (avgLen^2 + avgLen)}{2 * avgLen}, \tag{12}$$

where $actionSpace$ is number of possible actions and $avgLen$ is approximate average game length. For board size 13×13 alpha was $\alpha \approx 0.064$. If the noise leads to a selection of actions that are unfavorable it is overridden by the search algorithm. In the case a node is reencountered, these recommendations are taken into account in the action-selection step. Visit count is initialised $N(s, a) = 0$. The ratio of adding noise, ϵ I kept the same.

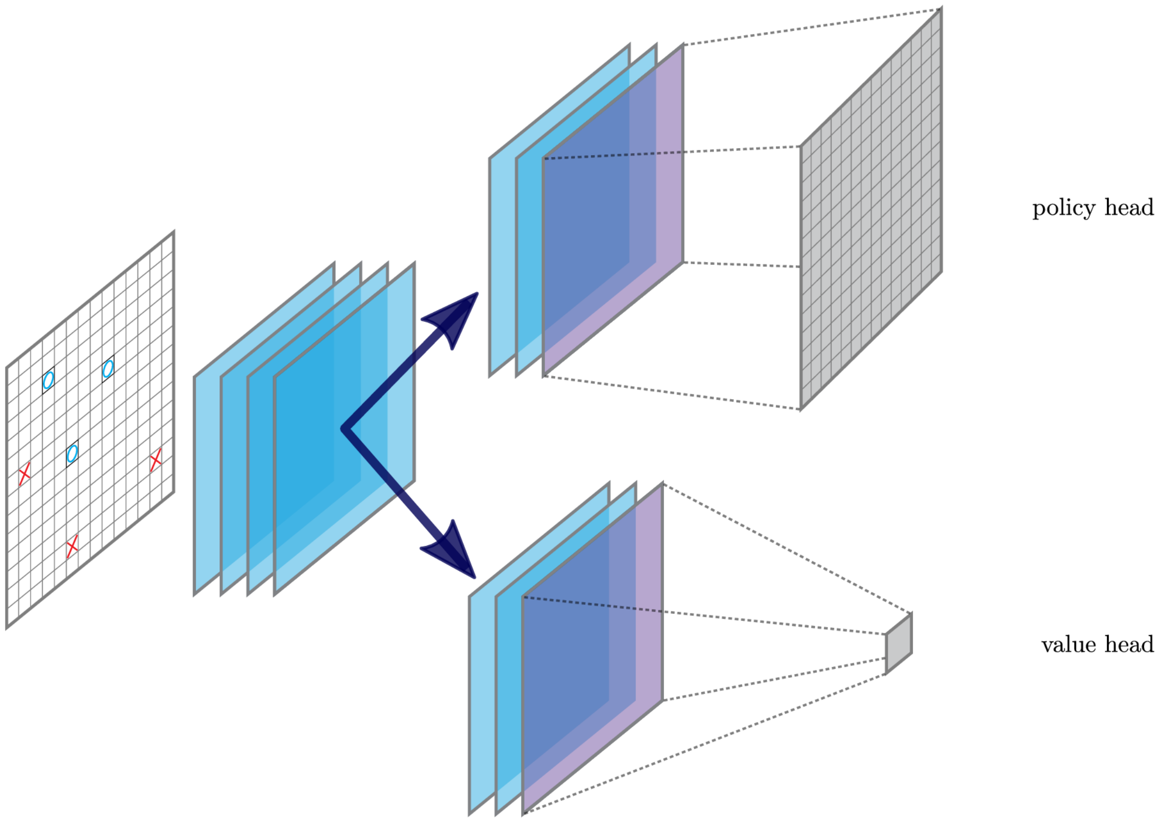

Figure 4.1: Architecture of my neural network with input on the left, four common convolutional layers dividing into two heads.

.

Constant C_PUCT For both simulation and evaluation I set this constant to 4, which is little bit lower than in the article [27].

Temperature τ I tried to used temperature $\tau = 1$ for the entire length of the game, however when I tried setting $\tau \rightarrow 0$ from 10th move for the remainder of the game. This might have helped speed up the training process a little bit.

4.3 Training

To speed up training I remove data from first generations I used growing sliding window [38], starting at 4 every two generations increased by 1 until it reaches its maximal size of 20. For optimization, I used algorithm Adam, with learning rate 0.001. The loss function for policy head was Kullback-Leiber divergence

$$KLDiv(p, \pi) = -\sum_{i=0}^{K} p_i \log(\frac{\pi_i}{p_i}), \tag{13}$$

where K is length of the input vectors. Value head was trained on results of the games using mean square error (MSE) loss

$$MSE(v, r) = -(v - r)^2. \tag{14}$$

I trained on batches of size 256. Each training iteration I did 3 epochs on data set which proved to be enough, because older data were used multiple times. When I was trying to increase number of epochs there was significant decrease in performance.

In game *Gomoku* positions are invariant to rotation and reflection. To augmented data I used dihedral rotations of positions, also I merged duplicated data by averaging policy vectors and results of the game.

Originally I wanted to do this additional step to make sure that the parameters are performing better than previous. But this step was too time-consuming and I decided to skip it.

Chapter 5

Implementation

For the implementation I combined a couple of technologies. I will divide my implementation into three separate parts, neural network, Monte Carlo Tree search and training logic, because I treated them individually and although they are tied together using different technologies. The algorithm itself takes a huge amount of resources during training. For this reason, I had to put a lot of effort into efficient implementation.

5.1 Training logic and definition of neural network

This part consists of self-play, evaluation, training, and definition of the neural network. Language *Python* [33] seemed to be the obvious choice for it. It has a lot of frameworks for machine learning and mathematical libraries. Machine learning framework I used for the implementation of my neural network is Pytorch [21], which provides a great number of features required for machine learning and I was already familiar with it. For other mathematical operations and for work with data I used *Numpy* library.

Training resources. For data generation I used resources on *MetaCentrum* in parallel, each process running on 8 CPUs for about half an hour. Created data set of about $20,000$ state samples were transferred to the central node with *GPU* where new parameters were optimized using multiple latest datasets as described in section 4.3. New parameters were then transferred back to *MetaCentrum* and new data set was generated with them 5.1.

5.2 Monte Carlo tree search.

This was the largest portion of my work and most time-consuming. My first attempt was to implement MCTS in Python language. Python was my first choice because of the availability of machine learning frameworks and language features enabling fast prototyping. This has proved too slow to be useful because Python is interpreted language, which has a detrimental effect on performance. Also, I run into limitations that Python has in multithreading, which was essential to the effective parallelization of MCTS.

C++ Python Extension. To make implementation more efficient I decided to rewrite MCTS into *c++*[11] programing language. Although this proved to be more difficult than I originally thought it would be, this effort paid off and the running it on CPU I achieved 4-times speed up. I suspect that the speed up would be even more significant when utilizing

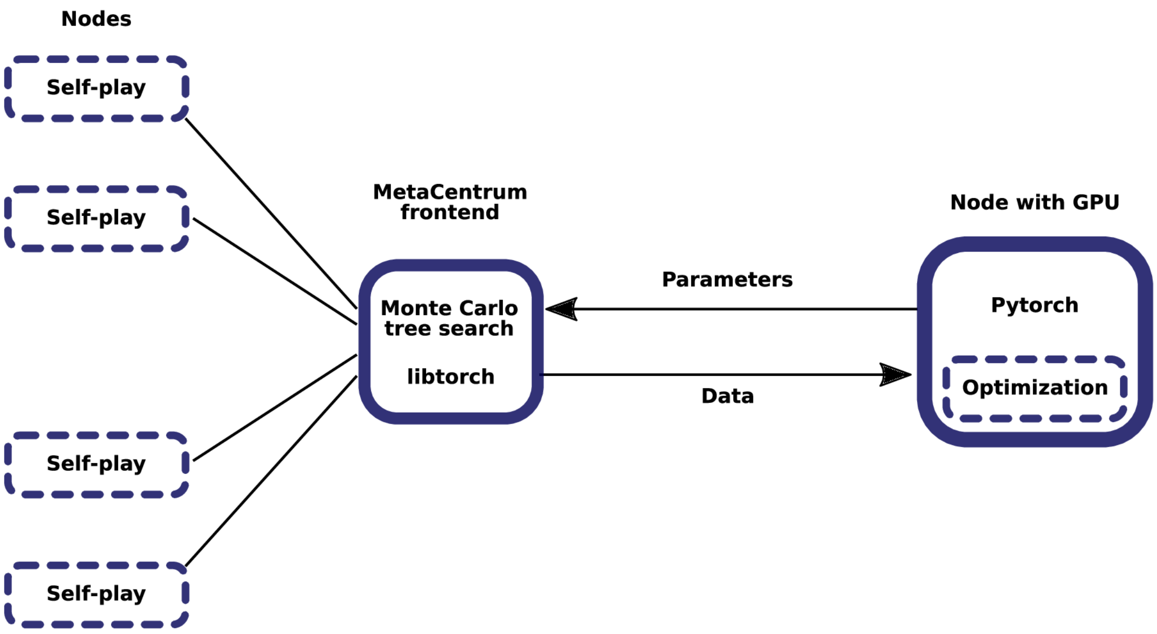

Figure 5.1: Training configuration.

GPU because of evaluation of the neural network took a significant portion of the time. To be able to keep parts of my previous work in python I created bindings to *Python* using *Pybind11* [12], which provides a very elegant way of using the c++ code in Python with features such as casting Python object to c++ objects back and forth but very importantly support for mathematical python library Numpy [20]. This also meant I was free of pythons Global Interpreter Lock restrictions and enabled me to implement parallelization with *virtual loss* 5.2.

My original idea was to use callbacks to Python for neural network evaluation. Although it is possible, it was too difficult to implement properly for the multithreaded application.

Eventually I used *Pytorch c++* interface, which enables define and use neural network in *c++*. There is also an option to define and train network in *Python* and using just in time compiler compile it and then load to *c++*. In my implementation I compiled the neural network and then loaded it with *c++* extension.

Using extension gave me the advantage of dynamical properties of *Python* and having almost all computation heavy code written in much faster *c++*.

Parallelization. To parallelize the Monte Carlo tree search there are a couple of approaches. Easiest to implement is tree parallelization, used in *Fuego* program playing *Go*, where we have multiple clones of the search tree and we run a simulation on them separately and then merge statistics. This, however, have a disadvantage to is and that is the quality of simulation is lower than running the same number of simulation on the same tree and is not very well scalable to a larger number of threads [25]. Another approach that was used by authors of the *AlphaZero* algorithm, and which I used is running simulation in the same tree. To prevent threads exploring the same parts of the tree, *virtual loss* has to be used. When a thread selects an action it decreases its total value, as it would if it leads to a loss. This makes the action less favorable for another thread visiting the same node. The additional overhead coming with searching the same tree in using locks is balanced out by better scalability to multiple threads and more single thread like results.

Build system. To build the extension I used a combination of *Python setuptools* and *cmake*. Using *setuptools* enables to install binaries in to correct place as any other python extension, while using *cmake* is very pleasant because of its feature of finding dependencies and using correct compilation flags, required to link to them. Dependencies required for build are: *Pytorch*, *Python*, *Pybind11* and *Gnu Scientific Library* which is used for Dirichlet noise generation.

Computing with GPU. Extension is ready to be used with GPU acceleration, however I had problems linking to *cudnn* library on *MetaCentrum*.

5.3 Portability

I have used and tested everything on the Linux operating system. However every library I used is also available for Windows and *cmake* used for building *c++* extension is a cross-platform tool, although it may require some tweaking. Only the platform dependent code is *bash* invocation script.

5.4 Gomoku tournament manager interface

To compare my solution to existing *Gomoku* playing Artificial Intelligence (AI) I created an interface to *Piskvork* [17] tournament manager. This application provides a graphical user interface for human interaction and also the management of tournaments for AIs. This program runs only for windows and requires AI to be a windows binary that communicates with standard input/output. To be able to communicate with this manager I created small client as windows binary which could be executed by *Piskvork* and was resending commands from *Piskvork* manager to *Python* server. The server then started tree search and send selected move back. The manager was then executed under Linux using *Wine*[4] which translates Windows API calls into POSIX calls.

5.5 Heuristic

For evaluation and testing purposes I created simple heuristic, which is part of the Monte Carlo tree search module and is used if parameters of the neural network are not supplied. This heuristic subsidizes neural network and provides for search algorithm both *policy* and *value*. It keeps the position value of each field of the board for each player separately and when a move is made it updates position values in directions where the potential winning sequence can occur accordingly to a number of already placed stones in a row. *Policy* is computed as a sum of position values of both players and *value* is the ratio of the sum of all values for one player to the sum of all values for the opponent.

Chapter 6

Experiments

6.1 Performance with regard to simulations

To evaluate how an increase in the number of Monte Carlo simulations effects game strength I used heuristic 5.5 as a substitution for neural network, which is much faster to evaluate. As expected, using more simulations improved had a positive effect on the strength of the player. As a reference in this comparison, I used 1000 simulations, which plays at a reasonable strength. This was compared to players using 500, 1200, 2000 and 5000 simulations for evaluation of each move. The heuristic I used was biased and was disadvantageous to the first player. In table 6.1 we can see that playing strength was growing proportionally to the number of simulations. Each player was creating its own search tree with identical configurations.

I found out that using *Dirichlet* noise had a negative effect on the Monte Carlo tree search and it reduced performance by a significant margin. Tree search using no noise at all against tree search adding noise as described in section 3.3 won every game when making first move and 80% of the games when playing second. This could be due to inappropriate constant α for the combination of heuristic and MCTS. However, using noise for training is important.

6.2 Training progress

To see how training progress I have chosen several generations of parameters created during training. Each pair of parameters were evaluated against each other. Simulations were executed in separate trees and configuration of tree search was the same for every player and for every game. The temperature was set to $\tau \rightarrow 0$ for the whole length of the games so moves would be selected deterministically, choosing the best move from the policy provided by MCTS. The result 6.1 show improvements in each subsequent generation, most of the newer generations were playing convincingly stronger than previous. All models achieved stronger results when making the first move.

Simulations	Moving first	Moving second	Total	Percentage
1000	16	24	40	-
500	13	29	42	42%
	21	37	58	
1200	12	43	55	55%
	7	38	45	
2000	18	43	61	61%
	7	32	39	
5000	26	45	71	71%
	5	24	29	

Table 6.1: Results of games played with Monte Carlo Tree Search and heuristics. The number of wins is on top and the number of losses is bellow.

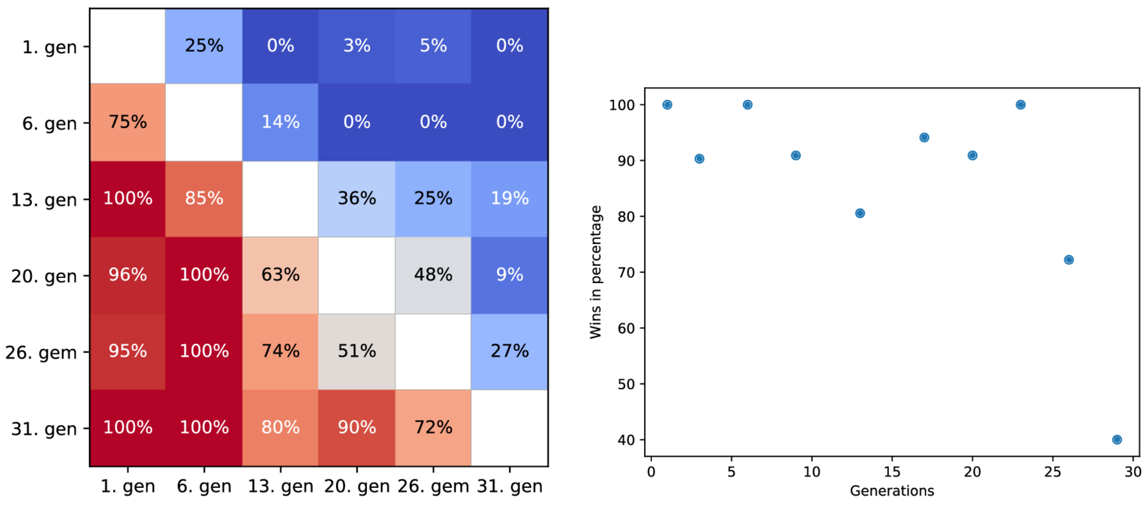

Figure 6.1: **Left.** Comparison of how well performed the latest parameters against the previous generation. **Right.** Compared selected parameters against each other.

6.3 Speed and scaling

Training the neural network using data from self-play is very computationally expensive. For every move thousand evaluations of the neural network are performed. Effective implementations are crucial to be able to train the network.

Because of a large number of loops and recursive nature of MCTS simulations python implementation was very slow and the problem grew even more on significance when the existing tree grew larger. To compare the efficiency of MCTS implementation I didn't use neural networks nor heuristic. Both implementations were running on a single thread without GPU acceleration.

The first comparison is how tree complexity affects the duration of tree search. The search starts from a clear state and measures how long does 1000 simulations take. The *c++* implementation is more than 30 times faster on the first 1000 simulations and its time complexity does not grow insignificantly and after reaching a certain point it stops growing completely. Python implementation, on the other hand, is very inefficient and after 5000 iterations it is nearly 10 times the original time. Changes in duration in relations to number of simulation can be seen in figure 6.2 and figure 6.3.

In figure 6.4 is comparison of duration of 1000 simulations with a different number of threads. Time efficiency grows quickly but after surpassing a number of available cores it plateaus.

6.4 Comparison to other programs

Comparison to existing programs playing *Gomoku* was done through tournament manager *Piskvork*. From previous data, a conclusion can be made that the training process works. However, it is hard to estimate how well because it might not reach its full potential. Unfortunately, due to the time it takes to fully train the networks, it was not able to defeat existing solutions. To be comparable to an existing solution, it would have to be trained for a far longer time. The neural network was not able to learn effective strategies and tree search did not perform enough simulations to make up for it. I compared to my neural network to program brain-Crush, which is ranked 45th in *Gomocup* ranking [16], because it is not using tree search at all or it is very limited and I could try, if given enough time my algorithm would be able to defeat it.

Games were played on board size 13×13 and both algorithms were given thinking time 20 seconds. In this time my algorithm was able to perform approximately 3000 simulations. Since the neural network was not able to provide good strategies for game openings, the search was not able to improve it enough to play reasonable sequences of moves. This resulted in blocking attempts of another player and eventually loss. Some game results can be seen in figure 6.5.

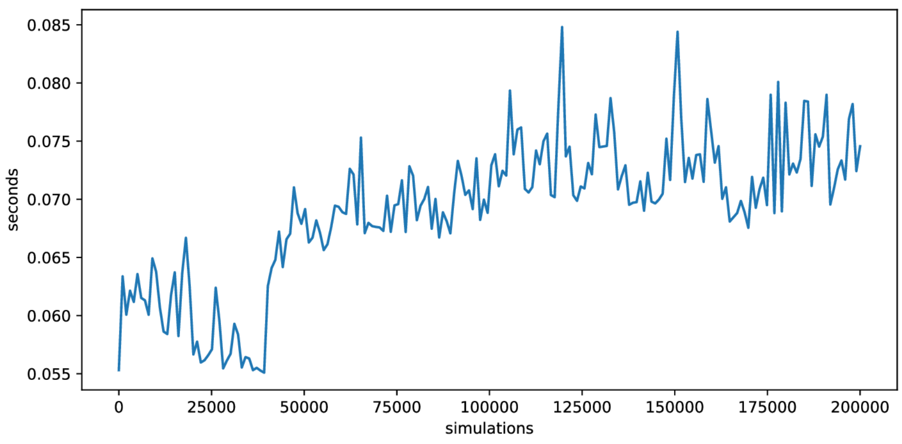

Figure 6.2: Duration of 1000 simulations in relation to number of simulations performed in the same search tree, *c++* python extension.

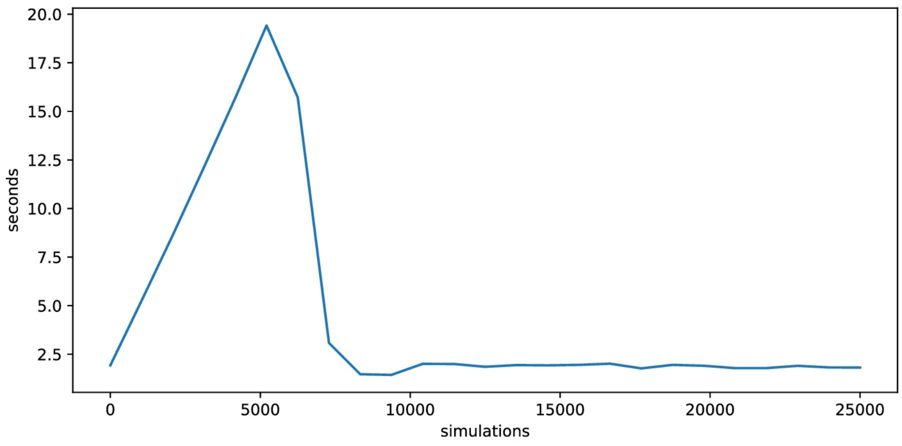

Figure 6.3: Duration of 1000 simulations in relation to number of simulations performed in the same search tree, python implementation.

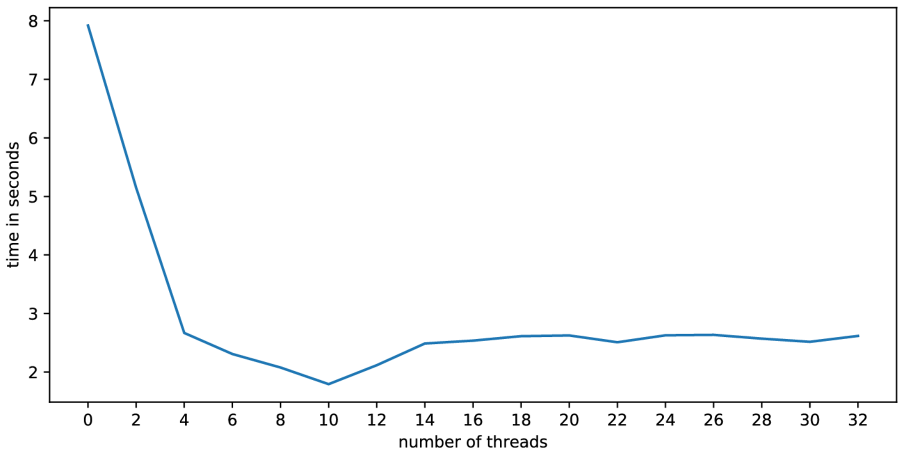

Figure 6.4: Effect of number of threads on duration of 1000 simulations on machine with 8 cores.

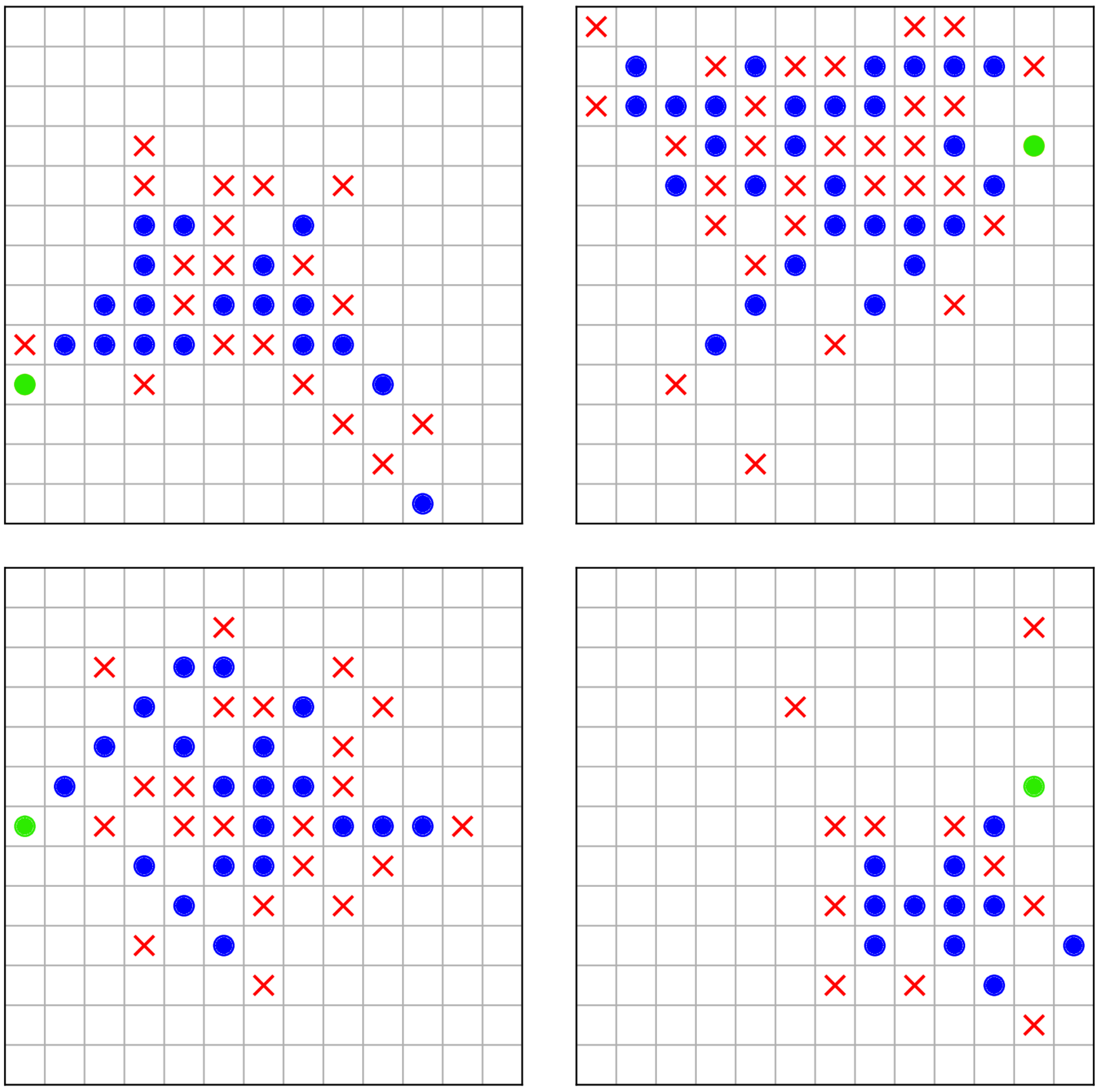
Figure 6.5: Terminal game positions from match between my algorithm (red) and brain-Crusher (blue). The winning move is colored green.

Chapter 7

Conclusion

The goal of this work was to create artificial intelligence that would be able to learn how to play game *Gomoku*. This goal was fulfilled only partially. From experiments, we can see that the strength of artificial intelligence was steadily growing and probably has not reached its full potential. On the contrary strength of the algorithm has not grown to expectations and achieved very poor results in games against other players.

I managed to create very efficient implementation, which provided a huge increase in speed compared to my previous attempts. This enabled me to generate data more efficiently. However, the training process requires much more computation then I used. During my training, I created 31 generations of parameters and overall generated more than 1 gigabyte of data.

I am planning to train it for longer to see what are the limits of my algorithm. I believe that if more time was spent on training, the algorithm will improve sufficiently to defeat some weaker AIs.

Bibliography

[1] Abramson, B.: Expected-outcome: A general model of static evaluation. *IEEE transactions on pattern analysis and machine intelligence.* vol. 12, no. 2. 1990: pp. 182–193.

[2] Anthony, M.: *Discrete Mathematics of Neural Networks: Selected Topics.* Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics. 2001. ISBN 9780898714807.

[3] Auer, P.; Cesa-Bianchi, N.; Fischer, P.: Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning.* vol. 47, no. 2. May 2002: pp. 235–256. ISSN 1573-0565. doi:10.1023/A:1013689704352.

[4] Bob Amstadt, A. J.: Compatibility layer. 1993. https://www.winehq.org.

[5] Campbell, M.; Hoane Jr, A. J.; Hsu, F.-h.: Deep blue. *Artificial intelligence.* vol. 134, no. 1-2. 2002: pp. 57–83.

[6] David, O. E.; Netanyahu, N. S.; Wolf, L.: Deepchess: End-to-end deep neural network for automatic learning in chess. In *International Conference on Artificial Neural Networks.* Springer. 2016. pp. 88–96.

[7] Edwards, D. J.; Hart, T.: The alpha-beta heuristic. *Technical Report 30, MIT.* 1963.

[8] Hahnloser, R. H.; Sarpeshkar, R.; Mahowald, M. A.; et al.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature.* vol. 405, no. 6789. 2000: pp. 947–951.

[9] He, K.; Zhang, X.; Ren, S.; et al.: Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016. pp. 770–778.

[10] Ioffe, S.; Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167.* 2015.

[11] ISO: *ISO/IEC 14882:1998: Programming languages — C++.* pub-ISO:adr: pub-ISO. September 1998. 732 pp.

[12] Jakob, W.; Rhinelander, J.; Moldovan, D.: pybind11 — Seamless operability between C++11 and Python. 2016. https://github.com/pybind/pybind11.

[13] Jouppi, N. P.; Young, C.; Patil, N.; et al.: In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA).* IEEE. 2017. pp. 1–12.

[14] Knuth, D. E.; Moore, R. W.: An analysis of alpha-beta pruning. *Artificial intelligence*. vol. 6, no. 4. 1975: pp. 293–326.

[15] Kocsis, L.; Szepesvári, C.: Bandit based monte-carlo planning. In *European conference on machine learning*. Springer. 2006. pp. 282–293.

[16] Lastovicka, P.: Gomocup Elo Ratings. [Online; accessed 14.05.2019]. Retrieved from: https://gomocup.org/elo-ratings/

[17] Lastovicka, P.: Piskvork manager. [Online; accessed 10.05.2019]. Retrieved from: https://gomocup.org/download-gomocup-manager/

[18] LeCun, Y.; Bengio, Y.; et al.: Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*. vol. 3361, no. 10. 1995: pp. 255–258.

[19] LeCun, Y.; Boser, B.; Denker, J. S.; et al.: Backpropagation applied to handwritten zip code recognition. *Neural computation*. vol. 1, no. 4. 1989: pp. 541–551.

[20] Oliphant, T.: *Guide to NumPy*. 01 2006.

[21] Paszke, A.; Gross, S.; Chintala, S.; et al.: Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*. vol. 6. 2017.

[22] Plaat, A.; Schaeffer, J.; Pijls, W.; et al.: A new paradigm for minimax search. *arXiv preprint arXiv:1404.1515*. 2014.

[23] Rosin, C. D.: Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*. vol. 61, no. 3. 2011: pp. 203–230.

[24] Rumelhart, D. E.; Hinton, G. E.; Williams, R. J.: Learning representations by back-propagating errors. *Nature*. vol. 323, no. 6088. 1986: pp. 533–536. ISSN 1476-4687. doi:10.1038/323533a0.

[25] Segal, R. B.: On the scalability of parallel UCT. In *International Conference on Computers and Games*. Springer. 2010. pp. 36–47.

[26] Shaun Brewer, R. B.; Kryukov, K.: CCRL 40/40 Rating List. [Online; accessed 04.05.2019]. Retrieved from: https://ccrl.chessdom.com/ccrl/4040/

[27] Silver, D.; Huang, A.; Maddison, C. J.; et al.: Mastering the game of Go with deep neural networks and tree search. *nature*. vol. 529, no. 7587. 2016: page 484.

[28] Silver, D.; Hubert, T.; Schrittwieser, J.; et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. vol. 362, no. 6419. 2018: pp. 1140–1144. ISSN 0036-8075. doi:10.1126/science.aar6404. https://science.sciencemag.org/content/362/6419/1140.full.pdf. Retrieved from: https://science.sciencemag.org/content/362/6419/1140

[29] Silver, D.; Schrittwieser, J.; Simonyan, K.; et al.: Mastering the game of Go without human knowledge. *Nature*. vol. 550, no. 7676. 2017: page 354.

[30] Sutton, R. S.; Barto, A. G.: *Reinforcement learning: An introduction*. MIT press. 2018.

[31] Takada, J.: 2017 World Computer Shogi Championship ranking. [Online; accessed 09.05.2019].
Retrieved from: http://www2.computer-shogi.org/wcsc29/index_e.html

[32] Tord Romstad, M. C.; Kiiski, J.: Stockfish: A strong open source chess engine. [Online; accessed 04.05.2019].
Retrieved from: https://stockfishchess.org/

[33] Van Rossum, G.; Drake, F. L.: *Python 3 Reference Manual*. Paramount, CA: CreateSpace. 2009. ISBN 1441412697, 9781441412690.

[34] Wedd, N.: Human-Computer Go Challenges. 2018. [Online; accessed 12.05.2019].
Retrieved from: http://www.computer-go.info/h-c/index.html

[35] Wikipedia contributors: Perfect information — Wikipedia, The Free Encyclopedia. 2019. [Online; accessed 13-May-2019].
Retrieved from: https://en.wikipedia.org/w/index.php?title=Perfect_information&oldid=876697251

[36] Wikipedia contributors: Shannon number — Wikipedia, The Free Encyclopedia. 2019. [Online; accessed 18.04.2019].
Retrieved from: https://en.wikipedia.org/w/index.php?title=Shannon_number&oldid=893877259

[37] Wikipedia contributors: Zero-sum game — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Zero-sum_game&oldid=885401216. 2019. [Online; accessed 03.05.2019].

[38] Young, A.: Lessons From AlphaZero. [Online; accessed 03.03.2019].
Retrieved from: https://medium.com/oracledevs/lessons-from-alpha-zero-part-6-hyperparameter-tuning-b1cfcbe4ca9a