

Mastering the game of Connect 4 through self-play

Julian Wandhoven

fgz

September 11, 2022

Abstract

Alpha Zero is an AI algorithm, that is capable of learning to play zero sum stated multiplayer games. These types of games include Go, Chess, Phi Sho and so forth. This is done by training a neural network and from data generated by a Monte Carlo Tree Search. This document also explains how neural networks work and a short explanation of the infrastructure around the AI to allow for playing on remote devices.

[\[4\]](#)[\[5\]](#)

Contents

1	Methods	4
1.1	Reinforcement Learning	4
1.2	Game	6
1.2.1	Game Board	7
1.2.2	Actions	8
1.3	MCTS	9
1.3.1	Evaluation Basis	11
1.3.2	Leaf Selection	12
1.3.3	Node Evaluation and Expansion	12
1.3.3.1	end game state $\mathcal{S}(n_{L_l}) \in \mathbb{G}_{done}$	13
1.3.3.2	not end game state $\mathcal{S}(n_{L_l}) \notin \mathbb{G}_{done}$	13
1.3.4	Backfill	14
1.4	Neural Network	14
1.4.1	Introduction to Neural Networks	14
1.4.1.1	Fully Connected Layer	15
1.4.1.2	Convolutional Layer	15
1.4.1.3	Activation Function	16
1.4.1.4	Training	20
1.4.2	Network used by AlphaZero	21
1.4.2.1	Neural Network input	21
1.4.2.2	Neural Network Architecture	22
1.4.2.3	Training	24
1.5	Data generation	24
1.5.1	Action selection	24
1.5.2	Memory	25
1.5.2.1	Memory update	25
1.5.2.2	Model Training	25
1.6	Model evaluation	25
2	Evaluation	26
2.1	Elo-rating	26
2.1.1	Relativity of the Elo-rating	28
2.2	Elo results	28

3	Servers and Clients	30
3.1	The Web Server	30
3.2	The Data Server	30
3.3	The AI Server	32
3.3.1	State Transmission	33
3.3.2	Action Selection	34
3.3.3	Action Transmission	34
3.4	Clients	34
3.4.1	Desktop Client	34
3.4.2	iOS Client	35
4	Further definitions	35
4.1	Set Exclusion	35
4.2	Hadamard product	35
4.3	Inner Product	36
4.3.1	Matrices	36
4.3.2	n -dimensional Tensors	36
4.4	Submatrix	36
4.5	Vectorization	37
4.5.1	Tensors	37
4.6	Vector Concatination	37
4.7	Future Assignment Operator	38
4.8	Sets	38
5	Examples	39
5.1	Mcts	39
6	Proofs	41
6.1	tanh derivative	41
6.2	Softmax Derivative	42

Alpha Zero is an algorithm published in 2018 by Google Deepmind as the generalization of AlphaGo Zero, an algorithm that learned to play the game of Go using only the rules of the game. In the generalized version, the same principles were applied to Chess and Shogi. Unlike previous algorithms such as StockFish or Elmo that use hand-crafted evaluation functions along with alpha-beta searches over a large search space, Alpha Zero uses no human knowledge. Rather, it generates all information through self-play. It has been shown to achieve superhuman performance in Chess, Shogi and Go. In this project the AI will be trained to play connect4. The entire algorithm is implemented in C++ to increase efficiency during the Monte Carlo Tree Search (MCTS). Furthermore, to allow for better performance when playing, all computations are handled via a server (my desktop). A secondary server has also been added to allow for easy re-routing of the main server and handle things like elo-ratings (see section 2.1 on page 26) for all agents. Additionally, I have added a short introduction on how neural networks work and how they are trained in section 1.4 on page 14.

1 Methods

The Alpha Zero algorithm is a reinforcement learning algorithm using two major parts: a) a *Monte Carlo tree search* (MCTS) that is guided by b) the *neural network* to improve performance. The agent (computer player) runs a certain amount of simulation games using its MCTS and neural network. At each step, the MCTS evaluates the most promising next states as given by the neural network's estimation. The MCTS, by simulating games starting from the current state, will improve the neural network's prediction for that state. At the end of each game, the winner is determined and used to update the neural network's estimation of who would win a game starting from a certain state.

1.1 Reinforcement Learning

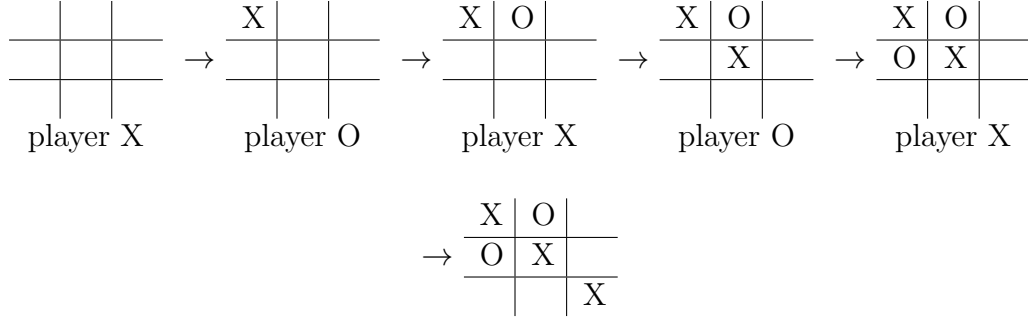
When training neural networks, there are three major possible situations: Supervised learning, unsupervised learning, and reinforcement learning. The first uses predetermined data with known in- and outputs the network is trained to predict. An example of supervised learning is the recognition of handwriting as the data is defined by humans. This method consists of

creating a large database of examples, and the neural network is then trained to predict a given output for all examples.

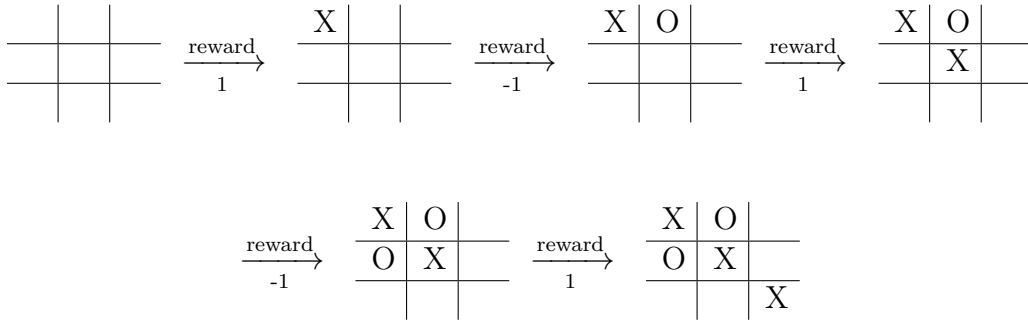
Unsupervised learning or self-organization is used when there is no previous available data and the neural network has to create those classifications itself. An example of unsupervised learning is Vector quantization. The algorithm sorts points in n -dimensional space into a predetermined amount of groups. Every group is defined by its centroid point. Training happens by selecting a sample point at random, and moving the closest centroid point towards the sample point by a fraction of the distance between them. The sample point is selected from the input data [6]. An example of both supervised and unsupervised learning can be seen in the demo¹.

These two methods represent the extreme ends of the spectrum. Reinforcement learning on the other hand can be thought of as an intermediate form. It uses a predetermined environment which gives positive, neutral and negative feedback. The neural network is then discouraged from taking actions leading to negative feedback and encouraged to take actions leading to positive feedback. The feedback is determined by the environment the agent learns to interact with. In this case, losing a game would be bad and result in negative feedback whereas winning a game leads to positive feedback. Ties lead to neutral feedback. The agent's learning is set up in such a way, that it is encouraged to take actions leading to positive feedback and discouraged from taking actions that lead to negative feedback. However actions, can lead to a loss that only occurs many game steps in the future. A common approach to solve this problem is to have the feedback propagate backwards to previous actions. In Alpha Zero, this is handled by the memory (see section 1.5.2 on page 25). When the game reaches an end state and a winner is determined, the feedback is propagated backwards up the game. If the player won, the feedback is positive. If he lost, it is negative. More specifically, if a player takes an action a_s at a state s , that leads to a win, the reward for that state is defined as $R(s, a_s) = 1$. On the other hand, if the action leads to a loss, the reward will be $R(s, a_s) = -1$. If the game ends in a tie, the reward is $R(s, a_s) = 0$. Every agent p will try to maximize $\sum_{s \in g \cap p} R(s, a_s)$. $g \cap p$ is the set of all states in which the player p takes an action. Let's look at a tic tac toe example of the following game:

¹demo is at <https://github.com/JulianWww/Matura-AlphaZero-demos>



Since player X won the game, the reward for every state $s \in g \cap X$ is $R(s, a_s) = 1$ and the reward for every state $s \in g \cap O$ is $R(s, a_s) = -1$. The reward for the entire game is:



The important thing to keep in mind is that reinforcement learning algorithms encourage actions that lead to a positive feedback and discourage actions that lead to a negative feedback.

1.2 Game

The game is the environment, that is used to train the AI. The game consists of constant unchanging game states. Every game state consists of a game board and a player. An end game state is a state at which the game is done. This means that one player won or the game ended in a tie. For connect4 this means four stones in a line or a full game board. Let \mathbb{G} be the set of all legal game states. Let \mathbb{G}_{done} be the set of all game states for which the game is done. At every gamestate one player is at turn. This means that that player

will take an action next. Let $\phi(s) : \mathbb{G} \rightarrow \{1, -1\}$ be the function mapping states to players. In the tick-tack-toe example from earlier we could say that:

$$\phi(s) = \begin{cases} 1 & s \in g \cap X \\ -1 & s \in g \cap O \end{cases} \quad (1)$$

More generally $\phi(s) = 1$ if the first player (the player that starts the game) is at turn and $\phi(s) = -1$ if the other player is a turn.

1.2.1 Game Board

Board games consist of placing stones of different types on a board with a certain amount of fields. Many games, like Go, Chess and Connect4, arrange their fields in a rectangular pattern. These games have two distinct stones. We can represent these game boards as stack of binary layers. Every layer is associated with one kind of stone. Each layer contains a one, where the board has a stone of the appropriate type and zeros everywhere else. For instance, the following tic tac toe game board can be represented by the following binary plane stack.

$$\begin{array}{|c|c|c|} \hline & X & O \\ \hline O & X & \\ \hline O & & X \\ \hline \end{array} \rightarrow \left[\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \right]$$

Internally, the game board is represented by a flat vector. The conversion from a game state $s \in \mathbb{G}$ to vector is defined as $vec(T_s(s))$. Where $T_s(s) : \mathbb{G} \rightarrow \mathbb{R}^{\dots}$ is the board's 3-dimensional board tensor. The vec function is

defined in section 4.5.1 on page 37.

$$vec \left(\left[\begin{bmatrix} a_{111} & \cdots & a_{11o} \\ \vdots & \ddots & \vdots \\ a_{1n1} & \cdots & a_{1no} \end{bmatrix} \cdots \begin{bmatrix} a_{m11} & \cdots & a_{m1o} \\ \vdots & \ddots & \vdots \\ a_{mn1} & \cdots & a_{mno} \end{bmatrix} \right] \right) = \begin{pmatrix} a_{111} \\ \vdots \\ a_{11o} \\ \vdots \\ a_{1n1} \\ \vdots \\ a_{1no} \\ \vdots \\ a_{m11} \\ \vdots \\ a_{m1o} \\ \vdots \\ a_{mn1} \\ \vdots \\ a_{mno} \end{pmatrix} \quad (2)$$

This operation for the tic tac toe board from before would look like this:

$$vec \left(T_s \left(\begin{array}{c|c|c} & X & O \\ \hline O & X & \\ \hline O & & X \end{array} \right) \right) = [010010001001100100]^T \quad (3)$$

1.2.2 Actions

Actions are numbers used to identify changes to the game. Every game has a set of all possible actions $\mathbb{A}_{possible} \subset \mathbb{N}_0$. In connect4, the set of all possible actions for the current player is $\mathbb{A}_{possible} = [0, 41]$. There is no need to have actions for the player, that is not at turn as these will never be taken. Every number is associated with a position on the game board. The mapping a to game fields is the following:

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41

Let $\mathbb{A}(s)$ be the set of all legal actions for a given state $s \in \mathbb{G}$. For all states $s_{done} \in \mathbb{G}_{done}$ the set of all legal actions $\mathbb{A}(s_{done})$ is the empty set. The function $\mathcal{A} : \mathbb{G} \times \mathbb{A} \rightarrow \mathbb{G}$ is used to get from one game state to another by taking an action. Where \mathbb{A} is the set of all legal actions the chosen game state. If we were to map action to position for tick tack toe as follows and that the game board is the following:

0	1	2	X	O	
3	4	5			
6	7	8			

State s

In this example player X is allowed to place a stone in any empty field $\mathbb{A}(s) = \{2, 3, 4, 5, 6, 7, 8\}$. Therefore $\mathcal{A}(a, s)$ is valid if $a \in \mathbb{A}(s)$ and otherwise invalid.

1.3 MCTS

A Monte Carlo tree search (MCTS) is a tree search algorithm that can be used to find sequences of actions leading to a desirable outcome. This is done by procedurally generating a directed graph of possible successor states to the current state or root state. In Alpha Zero it is used to improve the neural networks prediction.

A MCTS changes during simulation. MCTS simulation consists of three phases [Leaf Selection](#), [Node Evaluation and Expansion](#) and [Backfill](#).

This graph consists of nodes and edges. The edges represent connections between nodes while the nodes represent game states. Let $\mathbb{M}_{possible} = \{\mathcal{N}(s) \mid s \in \mathbb{G}\}$ be the set of all possible nodes, where $\mathcal{N} : \mathbb{G} \rightarrow \mathbb{M}_{possible}$ is a bijective function that maps a game state to a node. $\mathcal{S} : \mathbb{M}_{possible} \rightarrow \mathbb{G} = \mathcal{N}^{-1}$ is the inverse of \mathcal{N} . For notational simplicity, the set of all allowed actions for any node $n \in \mathbb{M}_{possible}$ is $\mathbb{A}(n) = \mathbb{A}(\mathcal{S}(n))$. The amount of

nodes and structure of the MCTS is changed by the algorithm itself during simulation. Therefore Let l be the index of the current simulation step. Let $\mathbb{L} = [0, S]$ be the set of all l , where S is a constant defined in the algorithms configuration file (section ?? on page ??). At the first step of the simulation, the Tree contains only the root node n_0 . The root node is the node is the starting point of the simulation. Let $\mathbb{M}_l \subseteq \mathbb{M}_{possible}$ be the set of all nodes in the tree at step $l \in \mathbb{L}$.

Every node $n_l \in \mathbb{M}_l$ has a set of edges $\mathbb{E}_l(n_l)$ at step l that connect it to other nodes. Let \mathbb{E}_l be the set of all edges in the current graph. $\mathcal{E}_l : \mathbb{M}_l \times \mathbb{A}_{possible} \rightarrow \mathbb{E}_l$ is a bijective function used to map nodes and actions to edges. Furthermore for $\mathcal{E}_l(n_l, a)$ to be valid a must be an element of $\mathbb{A}(n_l)$. The function $\mathcal{N}_{to_l} : \mathbb{E}_l \rightarrow \mathbb{M}_l$ maps an edge to the node it is pointing to while $\mathcal{N}_{from_l} : \mathbb{E}_l \rightarrow \mathbb{M}_l$ is used to find the node an edge is pointing from. Furthermore it is useful to distinguish expanded nodes from leaf nodes. Leaf nodes are nodes that don't have edges leading out of them. Let $\mathbb{M}_{leaf_l} \subseteq \mathbb{M}_l$ be the set of leafnodes. For every node $n_l \in \mathbb{M}_{leaf_l}$, the following is true by definition $\mathbb{E}_l(n_l) = \emptyset$. Expanded nodes are have edges leading out of them. Let $\mathbb{M}_{expanded_l} \subseteq \mathbb{M}_l$ be the set of expanded nodes. For every node $n_l \in \mathbb{M}_{expanded_l}$, the following is true by definition $\mathbb{E}_l(n_l) \neq \emptyset$.

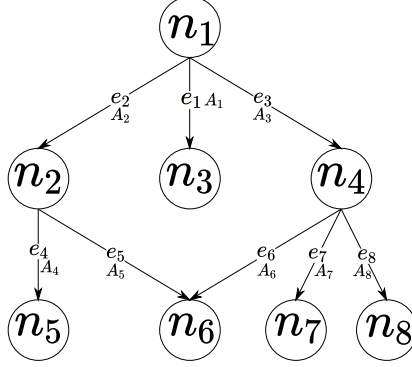


Figure 1: In the following small MCTS tree represents a possible graph at step l , where n_i are nodes, e_i are edges connecting the nodes and A_i are actions associated with the edge they are next to. The questionmark represents the indices. The set of all nodes $\mathbb{M}_l = \{n_1, n_2, \dots, n_8\}$ and $\mathbb{E}_l = \{e_1, e_2, \dots, e_8\}$. We can also see that $\mathbb{E}_l(n_1) = \{e_1, e_2, e_3\}$, $\mathbb{E}_l(n_2) = \{e_4, e_5\}$, $\mathbb{E}_l(n_3) = \emptyset$ and so forth. Because e_1 represents action A_1 taken at node n_1 , $\mathcal{E}_l(n_1, A_1) = e_1$. By the same logic $\mathcal{E}_l(n_2, A_2) = e_2$ and so forth. Because edge e_1 connects node n_1 to n_3 , $\mathcal{N}_{from_l}(e_1) = n_1$ and $\mathcal{N}_{to_l}(e_1) = n_3$. The same logic applies to all other nodes. The set of all expanded nodes $\mathbb{M}_{expanded_l} = \{n_1, n_2, n_4\}$ because they have at least one edge leading away from them. On the other hand, the set of all leaf nodes $\mathbb{M}_{leaf_l} = \{n_3, n_5, n_6, n_7, n_8\}$. Finally $\mathbb{M} = \mathbb{M}_{expanded} \cup \mathbb{M}_{leaf}$ (see section 5.1 on page 39 for all data)

1.3.1 Evaluation Basis

The MCTS's goal is to find a good estimations of the reward for a certain action at a certain state. This reward estimation is $Q_l : \mathbb{E}_l \rightarrow \mathbb{R}$. To define Q_l , the functions $W_l : \mathbb{E}_l \rightarrow \mathbb{R}$ and $N_l : \mathbb{E}_l \rightarrow \mathbb{N}_0$ are required. $N_l(e_l)$ is the amount of times an edge $e_l \in \mathbb{E}_l$ has been traversed. This means how many times σ (equation 6 on page 12) has chosen to follow the edge e_l to a new node. $W_l(e_l)$ is the sum of the reward computations from all $N(e_l)$ times the edge has been evaluated. With these two functions Q_l is defined as:

$$Q_l(e_l) = \begin{cases} 0 & N_l(e_l) = 0 \\ \frac{W_l(e_l)}{N_l(e_l)} & N_l(e_l) > 0 \end{cases} \quad (4)$$

The fourth and last of these functions is $P_l : \mathbb{E}_l \rightarrow \mathbb{R}$. P_l is the policy function, it's the neural network's preliminary estimation of how valuable the action is. To see what $P_l(e_l)$ is trained to approximate see equation 41 on page 24. This function is used to guide the search to more promising edges.

1.3.2 Leaf Selection

MCTS's evaluation starts by simulating future moves within the tree. This is done by selecting an edge and then following that edge to a new node. From there, the next edge and node are selected. This is repeated until a leaf node is reached. To select an edge and thus a node from the current node $n \in \mathbb{M}_l$ the function σ_l is used. To define σ_l we must first define the edge evaluation function $v_l : \mathbb{E}_l \rightarrow \mathbb{R}$. v_l is defined as follows:

$$v_l(e) = Q_l(e) + c_{puct} P_l(e) \cdot \frac{\sqrt{\sum_{b \in \mathbb{E}(\mathcal{N}_{from_l}(e))} N_l(b)}}{1 + N_l(e)} \quad (5)$$

Where $c_{puct} \in \mathbb{R}^+$ is the exploration constant used to define how important exploration is. The smaller c_{puct} is, more important Q and less important exploration and P . σ_l , for a given node $n \in \mathbb{M}_l$, is then defined as:

$$\sigma_l(n) = \mathcal{N}_{to_l}(\text{argmax}_l(\mathbb{E}_l(n))) \quad (6)$$

argmax_l returns the edge $e \in \mathbb{E}_l$ with the largest $v_l(e)$.

In order to find a leaf node n_{L_l} starting from the root node n_0 and be able to update the tree later on, the following algorithm is run L times, until a leaf node is found. First let $i \in [0, L]$ be the index of the iteration of this algorithm. Let $\mathbb{M}_{back,l,i} \subseteq \mathbb{M}$ be the set of nodes traversed by the algorithm at set i during simulation l . By definition $\mathbb{M}_{back,l,0} := \{n_0\}$. Than:

$$n_{L_i+1} = \sigma(n_{L_i}) \quad (7)$$

$$\mathbb{M}_{back,l,i+1} = \mathbb{M}_{back,l,i} \cup \{n_{L_i+1}\} \quad (8)$$

Once a leaf-node $n_{L_l} \in \mathbb{M}_{leaf_l}$ is found that node is evaluated. $\mathbb{M}_{back,l,L}$ will be important in section 1.3.4 on page 14.

1.3.3 Node Evaluation and Expansion

When a leaf node n_{L_l} is reached, there are two possible cases. Either $\mathcal{S}(n_{L_l})$ is an end game state or not.

1.3.3.1 end game state $\mathcal{S}(n_{L_l}) \in \mathbb{G}_{done}$

In this case the tree is not changed but backfill is still performed. The value $v_l \mathbb{R}_{-1,1}$ (66) of n_{L_l} is defined as if the current player $\phi(\mathcal{S}(n_{L_l}))$:

$$v_l = \begin{cases} 1 & \text{lost} \\ 0 & \text{tied} \\ -1 & \text{won} \end{cases} \quad (9)$$

This is done because the players alternate taking turns.

1.3.3.2 not end game state $\mathcal{S}(n_{L_l}) \notin \mathbb{G}_{done}$

In this case the nodes state is passed to the neural network (section 1.4 on page 14). The neural networks prediction function outputs a policy vector $\pi_l \in \mathbb{R}_{0,1}^{|\mathbb{A}|}$ (66) and a scalar $v_l \in \mathbb{R}_{-1,1}$ (66). The scalar v_l is the neural networks estimation of the expected reward at n_{L_l} . π_l is the estimation of the advantageousness of every action in \mathbb{A} . After evaluation, the leaf node is expanded. Let \mathbb{E}_{new} be the next set of all edges leading away from n_{L_l} . This set contains one edge for every action in $\mathbb{A}(n_{L_l})$. Let \mathbb{M}_{new} be the set of nodes, that can be reached in one game step from n_{L_l} . The various Tree sets and functions are then updated as follows for step $l + 1$.

$$\mathbb{M}_{l+1} = \mathbb{M}_l \cup \mathbb{M}_{new} \quad (10)$$

$$\mathbb{E}_{l+1} = \mathbb{E}_l \cup \mathbb{E}_{new} \quad (11)$$

$$\mathbb{E}_{l+1}(n) = \begin{cases} \mathbb{E}_{new} & n = n_{L_l} \\ \mathbb{E}_l(n) & \end{cases} \quad (12)$$

$$\mathcal{E}_{l+1}(n, a) = \begin{cases} \mathcal{E}_l(n, a) & n \neq n_{L_l} \\ e & \end{cases} \quad (13)$$

Where $e \in \mathbb{E}_{new}$ is the edge associated with action a at node n_{L_l} .

$$\mathcal{N}_{l+1}(e) = \begin{cases} \mathcal{N}_{l+1}(e) & e \in \mathbb{E}_l \\ n & \end{cases} \quad (14)$$

Where $n \in \mathbb{M}_{new}$ is the node, that edge e is pointing to.

$$\mathcal{N}_{from_{l+1}}(e) = \begin{cases} \mathcal{N}_{from_l}(e) & e \in \mathbb{E}_l \\ n_{L_l} & \end{cases} \quad (15)$$

$$\mathbb{M}_{expanded_{l+1}} = \{n \in \mathbb{M}_{l+1} : \mathbb{E}_{l+1}(n) \neq \emptyset\} \quad (16)$$

$$\mathbb{M}_{leaf_{l+1}} = \{n \in \mathbb{M}_{l+1} : \mathbb{E}_{l+1}(n) = \emptyset\} \quad (17)$$

$$P_{l+1}(\mathcal{E}_{l+1}(n, a)) = \begin{cases} P_l(\mathcal{E}_{l+1}(n, a)) & e \in \mathbb{E}_l \\ \pi_{l_a} & \end{cases} \quad (18)$$

1.3.4 Backfill

The value v_l is used to update the P , W and Q function as follows for every edge $e \in \mathbb{E}_{l+1}$:

$$N_{l+1}(e) = \begin{cases} 1 & e \notin \mathbb{E}_l \\ N_l(e) + 1 & \mathcal{N}_{to_{l+1}}(e) \in \mathbb{M}_{back,l,L} \wedge \mathcal{N}_{from_{l+1}}(e) \in \mathbb{M}_{back,l,L} \\ N_l(e) & \end{cases} \quad (19)$$

$$W_{l+1}(e) = \begin{cases} 0 & e \notin \mathbb{E}_l \\ W_l(e) + v_l & \mathcal{N}_{to_{l+1}}(e) \in \mathbb{M}_{back,l,L} \wedge \mathcal{N}_{from_{l+1}}(e) \in \mathbb{M}_{back,l,L} \\ W_l(e) & \end{cases} \quad (20)$$

Q_{l+1} is already defined according to (4). $\mathcal{N}_{to_{l+1}}(e) \in \mathbb{M}_{back,l,L} \wedge \mathcal{N}_{from_{l+1}}(e) \in \mathbb{M}_{back,l,L}$ checks if the edge e has been traversed during leaf selection. [1] In constrast to [4], this implementation stores the MCTS simulation values in the edges to allow for the fact that actions from two differant nodes leading to the same node are not identical and therefore should not be treated as such. But nodes of the same game state are identical and should be treated as such.

1.4 Neural Network

Search algorithms like MCTS are able to find advantageous action sequences. In game engines, the search algorithm is improved by using evaluation functions. These functions are generally created using human master knowledge. In the Alpha Zero algorithm, this evaluation function is a biheaded deep convolutional neural network trained by information gathered from the MCTS. In order to understand the training process, one must first understand how the neural network functions.

1.4.1 Introduction to Neural Networks

An artificial neural network or just neural network is a mathematical function inspired by biological brains. Although there are many types of neural networks, the only relevant one to this work is the feed forward network. These models consist of multiple linear computational layers separated by non-linear activation functions. Every layer takes the outputs of the previous layer, and applies a linear transformation to it [8]. There are many different feed-forward neural network layers and activation functions to choose from when designing a neural network. To focus this explanation, only the relevant ones will be discussed along with the back-propagation algorithm.

1.4.1.1 Fully Connected Layer

A fully connected layer is the most basic layer. It applies a simple matrix multiplication. The layer takes a $1 \times n$ dimensional matrix $x \in \mathbb{R}^{1 \times n}$ as an input and multiplies it by a weight matrix $w \in \mathbb{R}^{n \times m}$. This operation outputs a $1 \times m$ dimensional matrix to which a bias $b \in \mathbb{R}^{1 \times m}$ is added to form the output matrix $v \in \mathbb{R}^{1 \times m}$ containing the output values of the layer. v is then fed to the next layer. The addition of the bias vector b is optional. In some situations it is worth dropping the bias in favour of computational speed. The fully connected layer forward propagation function shall be defined as $\delta_{wb} : \mathbb{R}^n \rightarrow \mathbb{R}^m$

$$\delta_{wb}(x) = w \cdot x + b \quad (21)$$

1.4.1.2 Convolutional Layer

Convolutional layers are commonly used for image processing. They perform the same operations over the entire image searching for certain patterns. In order to achieve this, a set of kernels \mathbb{K} , of size $m \times n$, are defined for the layer. Kernels are similar to fully connected layers. They consist of a weight tensor $w \in \mathbb{R}^{m \times n \times l}$ and an optional bias scalar $b \in \mathbb{R}$. For every kernel $k \in \mathbb{K}$, the kernel's forward operation $\xi_k : \mathbb{R}^{m \times n \times l} \rightarrow \mathbb{R}$ is defined as:

$$\xi_k(i) = \langle w_k, i \rangle_I + b \quad (22)$$

where $\langle \rangle_I$ is the Tensor inner product defined in equation 59 on page 36. The convolutional operation $\Lambda : \mathbb{R}^{i \times j \times l} \rightarrow \mathbb{R}^{i-m+1 \times j-n+1 \times |\mathbb{K}|}$ is an element

wise operation. Given that $I \in \mathbb{R}^{i \times j \times l}$ is the layer input, every element of $\Lambda(I)_{abc}$ with $a \in [1, i - m + 1]$, $b \in [1, j - n + 1]$ and $c \in [1, |\mathbb{K}|]$ is defined as:

$$\Lambda(I)_{abc} = \xi_{k_c}(\langle I \rangle_{S_{m \times n, ab}}) \quad (23)$$

The submatrix indexing operation $\langle \rangle_s$ is defined in section 4.4 on page 36. For example given the following input tensor $I \in \mathbb{R}^{4 \times 4 \times 1}$:

$$I = \begin{bmatrix} [3] & [0] & [1] & [5] \\ [2] & [6] & [2] & [4] \\ [2] & [4] & [1] & [0] \\ [3] & [0] & [1] & [5] \end{bmatrix}$$

and the following kernel weight matrix $w_k \in \mathbb{R}^{3 \times 3 \times 1}$ along with the scalar $b \in \mathbb{R}$,

$$w_k = \begin{bmatrix} [-1] & [0] & [1] \\ [-2] & [0] & [2] \\ [-1] & [0] & [1] \end{bmatrix}$$

$$b = 7$$

there are four possible locations in which w_k can be placed within I . As there is only one kernel, the length of the set of all kernels $|\mathbb{K}| = 1$. This also means that $\Lambda(I) \in \mathbb{R}^{2 \times 2 \times 1}$. To calculate $\Lambda(I)_{111}$, we compute the kernel operation $\xi_k(I[[1, 3], [1, 3], \{1\}])$

$$\begin{aligned} \Lambda_{111} \left(\begin{bmatrix} [3] & [0] & [1] & [5] \\ [2] & [6] & [2] & [4] \\ [2] & [4] & [1] & [0] \\ [3] & [0] & [1] & [5] \end{bmatrix} \right) &= \begin{bmatrix} [3] & [0] & [1] \\ [2] & [6] & [2] \\ [2] & [4] & [1] \end{bmatrix} \circ \begin{bmatrix} [-1] & [0] & [1] \\ [-2] & [0] & [2] \\ [-1] & [0] & [1] \end{bmatrix} + 7 \quad (24) \\ &= -1 \cdot 3 + 0 \cdot 0 + 1 \cdot 1 - 2 \cdot 2 + 0 \cdot 6 + 2 \cdot 2 - 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 1 + 7 \\ &= 4 \end{aligned}$$

The same is done for $\Lambda(I)_{121}$, $\Lambda(I)_{211}$ and $\Lambda(I)_{221}$. This leads to a $\Lambda(I)$ of:

$$\Lambda(I) = \begin{bmatrix} [4] & [3] \\ [4] & [2] \end{bmatrix}$$

1.4.1.3 Activation Function

All neural network layers are linear functions. Thus, given two layer evaluation functions $f_1(x) = ax + b$ and $f_2(x) = cx + d$, the chained function $f(x) = f_1(f_2(x))$ is also linear because:

$$f(x) = f_1(f_2(x)) = a(cx + d) + b = acx + ad + b = ex + g \quad (25)$$

where a, b, c, d, e and $g \in \mathbb{R}$. In order to represent non linear functions, a non linear activation function f_a is added between two neural network layers. Thus, the chained function becomes $c(x) = f_1(f_a(f_2(x)))$. In this neural network, three different activation functions are used: *tanh*, *softmax*, and *LeakyReLU*. These functions are defined as follows:

tanh:

$\mathbb{R} \rightarrow \mathbb{R}$

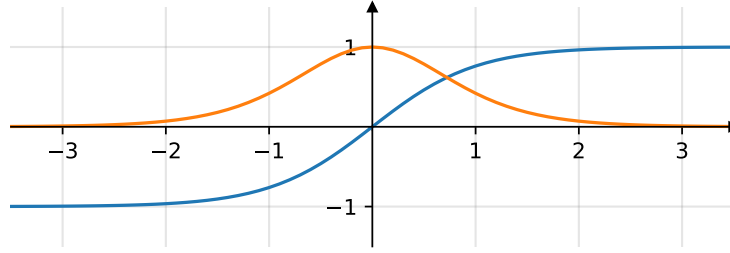


Figure 2: tanh function in blue and the tanh's derivative is in orange

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (26)$$

$$\frac{d}{dx} \tanh(x) = \text{sech}(x)^2 \quad (27)$$

see section 6.1 on page 41 for proof.

softmax:

$$\mathbb{R}^n \rightarrow \mathbb{R}^n$$

For a given input vector $v \in \mathbb{R}^n$. The output vector $o \in \mathbb{R}^n$ at every position $i \in [1, n]$ is:

$$o_i = \text{softmax}(v)_i = \frac{e^{v_i}}{\sum_{j \in v} e^j} \quad (28)$$

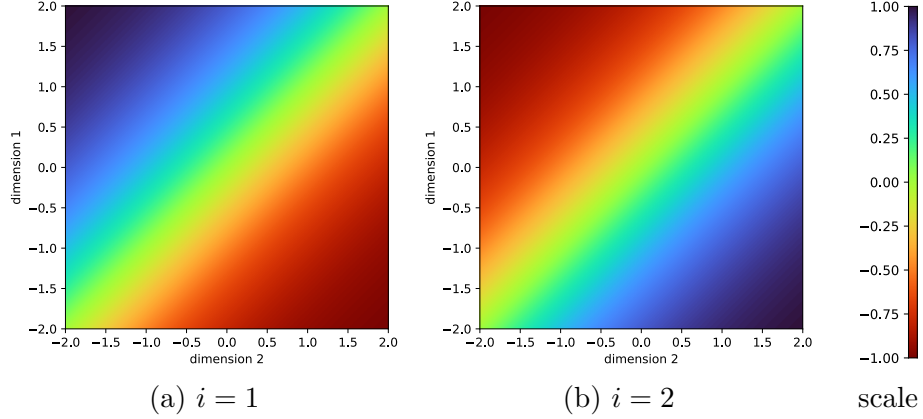


Figure 3: Graph of the softmax function from $\mathbb{R}^2 \rightarrow \mathbb{R}^2$. i is the index of the output dimension. Therefore, $i = 1$ refers to the output's first dimension and $i = 2$ refers to its second dimension.

Because the function's in- and outputs are n dimensional vectors, the derivative is an $n \times n$ dimensional matrix. When taking its derivative, $\frac{d}{dv_j} \text{softmax}(v)_i$, there are two possible cases.

case $j = i$:

$$\frac{d}{dv_j} \left[\frac{e^{v_i}}{\sum_{b \in v} e^b} \right] = \text{softmax}(v)_i \cdot (1 - \text{softmax}(v)_j) \quad (29)$$

case $j \neq i$:

$$\frac{d}{dv_j} \left[\frac{e^{v_i}}{\sum_{b \in v} e^b} \right] = -\text{softmax}(v)_i \cdot \text{softmax}(v)_j \quad (30)$$

Therefore, the derivative of the softmax function is:

$$\text{softmax}'(v)_{ij} = \begin{cases} \text{softmax}(v)_i \cdot (1 - \text{softmax}(v)_j) & i = j \\ -\text{softmax}(v)_i \cdot \text{softmax}(v)_j & i \neq j \end{cases} \quad (31)$$

LeakyReLU:

$\mathbb{R} \rightarrow \mathbb{R}$

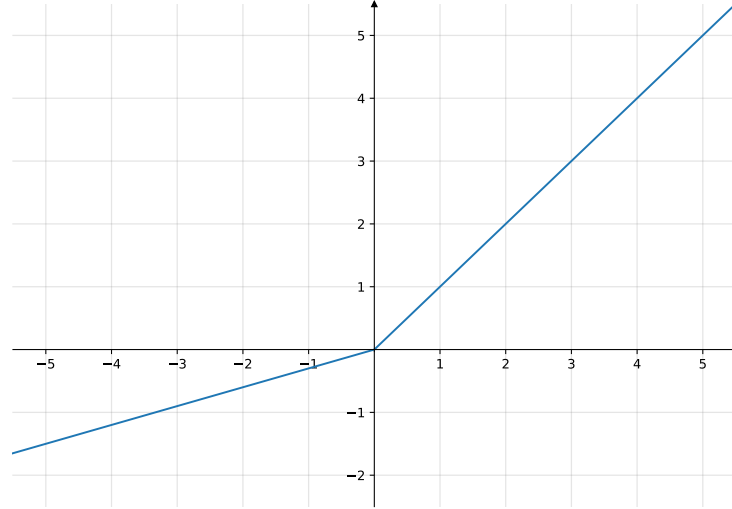


Figure 4: LeakyReLU with $c = 0.3$

$$\text{LeakyReLU}(x) = \begin{cases} x & x \geq 0 \\ x \cdot c & x < 0 \end{cases} \quad (32)$$

$$\frac{d}{dx}\text{LeakyReLU}(x) = \begin{cases} \frac{d}{dx}x & x \geq 0 \\ \frac{d}{dx}c \cdot x & x < 0 \end{cases}$$

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & x > 0 \\ c & x < 0 \end{cases} \quad (33)$$

where c is a constant describing the slope of the function for negative input values. The derivative of the LeakyReLU function is undefined for $x = 0$. However as we will be performing gradient descent on these functions the derivative must be defined for all $x \in \mathbb{R}$. A possible definition that accomplishes the objective is:

$$g(x) = \begin{cases} 1 & x \geq 0 \\ c & x < 0 \end{cases} \quad (34)$$

1.4.1.4 Training

Neural network training can be mathematically expressed as minimizing a loss function ℓ describing how inaccurate the network is. In our case, ℓ takes the neural network's predicted value vector Y_{pred} and the correct value vector Y_{true} . Y_{true} must be known before the computation begins. In AlphaZero, Y_{true} is generated by the MCTS. As with the activation, function there are many different possible loss functions. In this implementation, the mean-square-error(*mse*) loss function is used. $mse : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ is defined as:

$$\ell = \frac{|Y_{pred} - Y_{true}|^2}{n} \quad (35)$$

The network then performs gradient descent to find parameters that minimize ℓ . To make this introduction easier, I will use a fully connected neural network. For every layer in the network, starting with the last one, it must be determined in which direction and by how much the output values $Y_{pred_i} \in Y_{pred}$ should be "moved" to minimize ℓ . This change is described by ΔY_j , where j is the index of the last layer. Then, the change in the inputs to the activation function f_a must be computed using the saved activation function inputs A . ΔA will describe the change to A .

$$\Delta A = f'_a(A) \circ \Delta Y_j \quad (36)$$

The hadamard product \circ is defined in section 4.2 on page 35.

Next comes the update to the weight matrix w . Let Δw describe the change to w and let X be the input vector of the layer. Δw is than defined as:

$$\Delta w = \Delta A \cdot X^T \quad (37)$$

The layer's bias vector is updated in the direction of ΔA :

$$\Delta b \sim \Delta A \quad (38)$$

Lastly, the change to the output of the previous layer ΔY_{j-1} is computed.

$$\Delta Y_{j-1} = \Delta A \cdot w^T \quad (39)$$

This process is repeated until the foremost layer of the neural network is reached. This layer has the index $j = 0$.

1.4.2 Network used by AlphaZero

The neural network in Alpha Zero is used to estimate the value v and policy p for any game state or node n . v is the neural network's estimation of the state's expected reward. The policy $p \in \mathbb{R}^{|\mathbb{A}|}$ of a game state n represents the advantageousness of every action $a \in \mathbb{A}$, as estimated by the neural network.

1.4.2.1 Neural Network input

The neural network input is a game state or node n represented by two 7 x 6 binary images stacked on top of each other. One image X represents the stones belonging to the current player. While the second image Y represents the stones belonging to the other player. In both images, the pixel values are one where a stone belonging to the player they represent is located and zero if the field is empty or a stone belonging to the other player is located there. X and Y are then stacked on top of each other in the third dimension to form the input tensor $i_n = [X, Y] \in \mathbb{R}^{7 \times 6 \times 2}$. Consider the following Connect4 board (fig 5 on page 21).

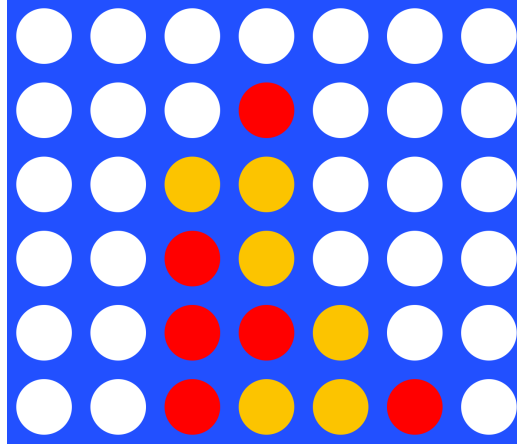


Figure 5

If red is the current player then:

$$X = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcolor{red}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{orange}{0} & \textcolor{orange}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{orange}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{red}{1} & \textcolor{orange}{0} & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{orange}{0} & \textcolor{orange}{0} & \textcolor{red}{1} & 0 \end{bmatrix}$$

$$Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcolor{red}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{orange}{1} & \textcolor{orange}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{orange}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{red}{0} & \textcolor{orange}{1} & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{orange}{1} & \textcolor{orange}{1} & \textcolor{red}{0} & 0 \end{bmatrix}$$

For clarification, the numbers are coloured in the same colour as the stones at that position. After stacking X and Y , i_n is:

$$i_n = \begin{bmatrix} [0, 0] & [0, 0] & [0, 0] & [0, 0] & [0, 0] & [0, 0] & [0, 0] \\ [0, 0] & [0, 0] & [0, 0] & \textcolor{red}{[1, 0]} & [0, 0] & [0, 0] & [0, 0] \\ [0, 0] & [0, 0] & \textcolor{orange}{[0, 1]} & \textcolor{orange}{[0, 1]} & [0, 0] & [0, 0] & [0, 0] \\ [0, 0] & [0, 0] & \textcolor{red}{[1, 0]} & \textcolor{orange}{[0, 1]} & [0, 0] & [0, 0] & [0, 0] \\ [0, 0] & [0, 0] & \textcolor{red}{[1, 0]} & \textcolor{red}{[1, 0]} & \textcolor{orange}{[0, 1]} & [0, 0] & [0, 0] \\ [0, 0] & [0, 0] & \textcolor{red}{[1, 0]} & \textcolor{orange}{[0, 1]} & \textcolor{orange}{[0, 1]} & \textcolor{red}{[1, 0]} & [0, 0] \end{bmatrix}$$

1.4.2.2 Neural Network Architecture

The neural network used by Alpha Zero consists of three main sub-modules, namely the residual tower, the value head and the policy head. The residual tower's purpose is to preprocess the data for the two heads. The value head determines the value v from the output of the residual tower. While the policy head computes the policy p . The residual tower consists of a convolutional block followed by six residual blocks.

The convolutional block consists of the following:

1. A convolutional layer consisting of 75 filters with a kernel size of 3 x 3
2. Batch normalization [\[3\]](#)
3. A non-linear rectifier (LeakyReLU).

Every residual block consists of the following modules:

1. A convolutional layer consisting of 75 filters with a kernel size of 3 x 3
2. Batch normalization [3]
3. A non-linear rectifier (LeakyReLU)
4. A convolutional layer consisting of 75 filters with a kernel size of 3 x 3
5. Batch normalization [3]
6. Batch normalization outputs are added to the block's input.
7. A non-linear rectifier (LeakyReLU)

Outputs are then passed to the value and policy head of the network for further evaluation. The value head consists of the following modules:

1. A convolutional layer consisting of 10 filters with a kernel size of 1 x 1
2. A fully connected layer of size 210
3. A non-linear rectifier (LeakyReLU)
4. A fully connected layer of size 1
5. \tanh activation function

The policy head consists of the following modules:

1. A convolutional layer consisting of 2 filters with a kernel size of 1 x 1
2. A fully connected layer of size 1

The output of the policy head p_{pre} is then masked with the allowed actions to form p_{masked} in such a way that p_{masked} is -1000 for all non-allowed actions. Finally, p_{masked} is passed through the softmax function to form π :

$$\pi = \text{softmax}(p_{masked}) \quad (40)$$

1.4.2.3 Training

Training is performed in batches of 256 states. The value head is updated using mean square error. The policy head is updated using mean square error as well. However all non-legal actions are ignored. This avoids unnecessary updating of the neural network. The value, the neural network is trained to predict for a certain MCTS node n , is equivalent to 1 if the player who took an action at node n did win, -1 if that player did lose and 0 if the game ended in a tie. The policy p_{a_l} to train for, for a given legal action $a_l \in \mathbb{A}(n)$ is:

$$p_{a_l} = \frac{N(n, a_l)}{\sum_{a \in \mathbb{A}(n)} N(n, a)} \quad (41)$$

For non legal actions $a_n \in (\mathbb{A}_{possible} - \mathbb{A}(n))$, p_{a_n} is defined as:

$$p_{a_n} = p_{pre_{a_n}} \quad (42)$$

1.5 Data generation

The data used to train the neural network is generated by letting the best agent play several games against itself, until enough data has been generated to allow for training. In every game, at every game state, the MCTS performs 50 simulations. Once the simulations are done the action is chosen.

1.5.1 Action selection

There are two methods for action selection for a given node n_t : deterministic and probabilistic. The first will always return the action $a = \text{argmax}(N(\mathcal{E}(n_t, a \in \mathbb{A}(n_t))))$ of the most traversed edge, while the second will return a random action where the probability of selecting an action $a_i \in \mathbb{A}(n_t)$ is:

$$P(X = a_i, n_t) = \frac{N(\mathcal{E}(n_t, a_i))}{\sum_{j \in \mathbb{A}(n_t)} N(\mathcal{E}(n_t, j))} \quad (43)$$

($\mathbb{A}(s)$ are the allowed actions for state s .) Action selection during the training phase shall initially be probabilistic, and deterministic later on. The handover point shall be defined as the configurational constant 'probabilistic_moves' $\in \mathbb{N}^+$. During games outside the training loop, actions are always selected deterministically.

1.5.2 Memory

The memory stores a certain amount of memory elements. A memory element consists of a gamestate $g \in \mathbb{G}$, its action values $v \in \mathbb{R}^{|\mathbb{A}|}$ and the true reward $r \in \{1, -1, 0\} = R(g, a)$ where a is the action taken during play at that game state. The memory stores memory elements in a long list. After an action has been selected, but before any updates to the game simulation are made, the current game state is passed to temporary memory along with its action values v . Together they create a new memory element. This element's r is currently undefined. v is defined as:

$$v_a = \begin{cases} P(X = a, \mathcal{N}(g)) & a \in \mathbb{A}(g) \\ p_{pre_a} & \end{cases} \quad (44)$$

$\mathbb{A}(g)$ is the set of all legal actions. p_{pre} is defined in section ?? on page ??, and is used for all non legal actions. P is defined in equation 43 on page 24.

1.5.2.1 Memory update

Once the game is over, the winning player is determined and the value r of every memory element in the temporary memory is updated. r is 1 if the player taking an action at that state won, -1 if he lost and 0 if the game ended in a draw. The updated states are then passed to memory.

1.5.2.2 Model Training

Once the memory size exceeds 30'000 states, the self-playing stops and the neural network is trained as described in section: 1.4.2.3.

1.6 Model evaluation

In order to train the neural network, the "best player" generates data used to train the current network. After every time the current neural network has been updated, it plays 20 games against the best player. If it wins more than 1.3 times as often as the current best player, it is considered better. If this is the case, the neural network of the "current player" is saved to file and the old "best player" is replaced with the "current player" to become the new "best player". It is advantageous to force the network to win 1.3 times as often as that reduces the chance of the network just getting lucky.

2 Evaluation

To give us an idea of how good a player is, it would be useful to express performance using a single number. This number should not only give us a ranking but also allow for predictions of the winner of a game between two players and thus give us a measure of the relative strength of the players. One such rating method is the so called elo-rating method. [2]

2.1 Elo-rating

The elo-rating system assigns every player p a number $r_p \in \mathbb{R}$. In general, the larger r_p the better the player. More specifically, given two players a and b with elo-ratings r_a and r_b , the expected chance E of a winning against b is [4]:

$$E = \frac{1}{1 + e^{(r_b - r_a)/400}} \quad (45)$$

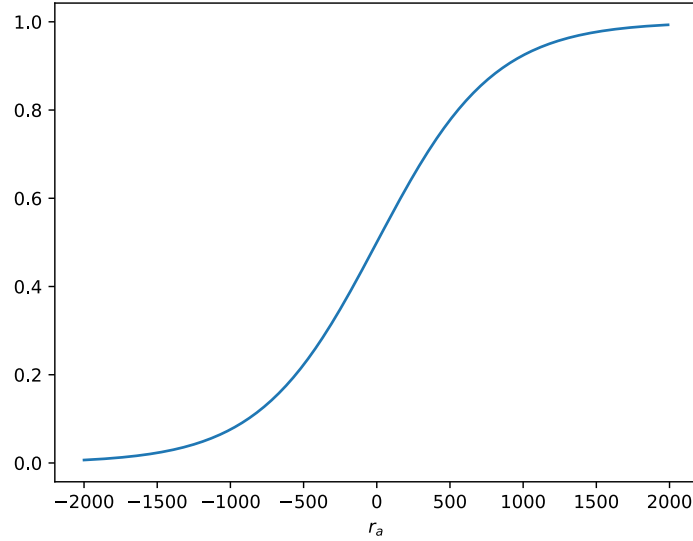


Figure 6: elo-rating win probability for $r_b = 0$

This function describes a sigmoid curve. This makes sense, because if the players have major strength discrepancies E converges to 1 or 0. When a and b play a game against each other, their elo,ratings are updated as follows[2]:

$$r_{n+1} = r_n + K(W - E) \quad (46)$$

with:

r_{n+1} the new rating for the player.

r_n the current rating of the player.

$W = s_a$ which is defined by equation 47 where a is the player to be updated.

E the expected chance of winning, see equation 45.

K is a constant controlling the sensitivity of the update function.

However, to avoid slow convergence of elo-ratings, a more direct formula is used to approximate the rating of an agent a . This is done by playing a predetermined amount of games against player b whose elo-rating r_b is known and unchanged throughout this process. First, a and b play a predetermined amount of games m and the score s_a of a is computed as [2]:

$$s_a = \frac{1}{m} \sum \begin{cases} 1 & a \text{ wins} \\ \frac{1}{2} & \text{tie} \\ 0 & a \text{ loses} \end{cases} \quad (47)$$

Assuming that this is the probability of a winning against b , a 's elo-rating can be computed by solving equation 45 to r_a (fig 7 on page 28):

$$r_a = r_b - \ln \left(\frac{1 - s_a}{s_a} \right) \cdot 400 \quad (48)$$

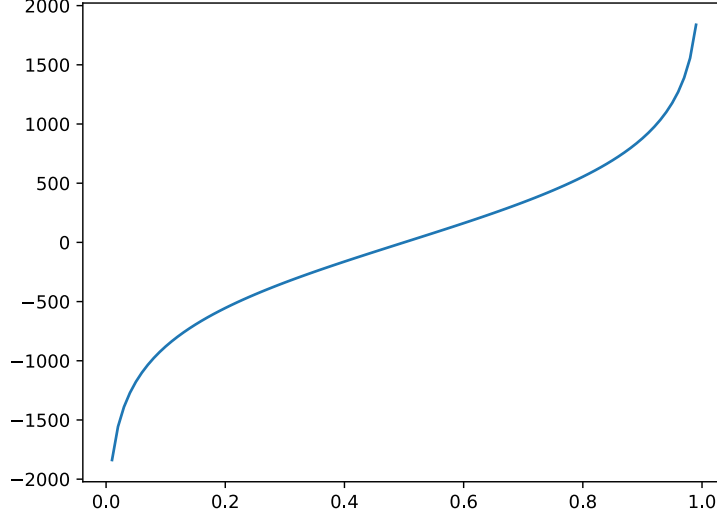


Figure 7: elo inverse function

Since a ranking of all the agents already exists (see section 1.6 on page 25), an agent's elo-rating can be computed by playing against an older version and then using equation 48 to determine its elo-rating.

2.1.1 Relativity of the Elo-rating

The only problem is that elo is a relative rating. The rating of any other agent depends on its performance against other agents and their elo-ratings. Therefore, one must give the system a base rating for at least one predefined agent. In this case, there are no previously known elo-rated agents, so I defined the untrained agent's elo-rating as 100. All other elo-ratings are relative to that.

2.2 Elo results

The rating r_i of any agent version i must in general be greater than the rating of the last version $r_i > r_{i-1}$. Furthermore, the expected minimal increase in rating $\Delta r_{min} = r_i - r_{i-1}$ is:

$$\Delta r_{min} = -\ln\left(\frac{1 - s_i}{s_i}\right) \cdot 400 \quad (49)$$

As a certain scoring threshold $\theta = 1.3$ was used during training to minimize the effect of noise in the evaluation, a prediction of s_i can be made. Given that s_a and s_b are the scores of two players that play against each other, then by definition:

$$s_a + s_b = 1 \quad (50)$$

Due to the imposed scoring threshold θ and the assumption that there are no ties:

$$s_a \geq s_b \cdot \theta \quad (51)$$

(if s_a has a higher version number than s_b)

For $\theta = 1.3$ this means that the expected average change in rating Δr :

$$\Delta r \geq -\ln\left(\frac{1}{\theta}\right) \cdot 400 = \Delta r_{min} \cong 105 \quad (52)$$

Collected data shows this to be true (fig 8 on page 30). The same data shows that the average Δr is in fact roughly 408, which would equate to a θ of

$$\theta = \frac{1}{e^{\frac{-\Delta E}{400}}} \cong 2.8 \quad (53)$$

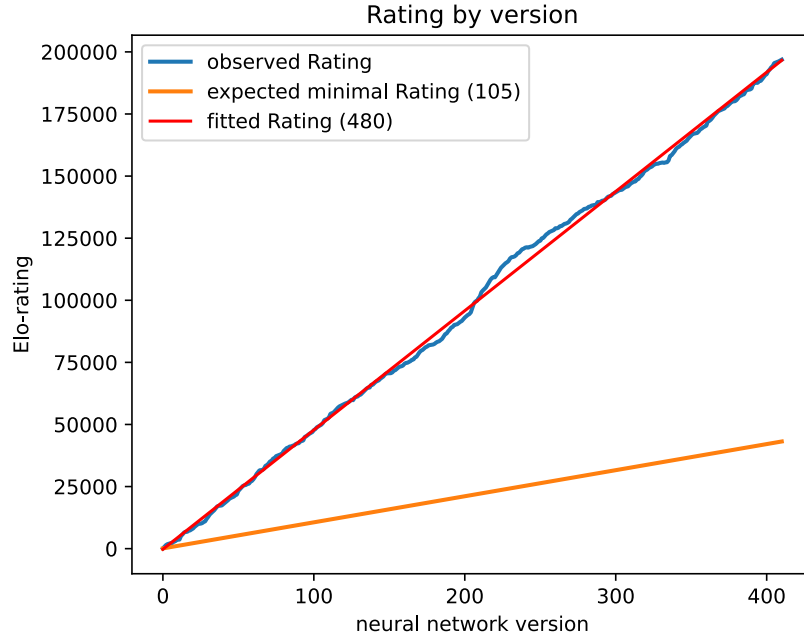


Figure 8: Elo-rating of agents based on their version along with the expected minimal rating Δr_{min} and the best fitted rating Δr .

3 Servers and Clients

In computer science, server-client-communications are a form of distributed application, that allows multiple machines to communicate and share data. In general, the server will wait for connections, while the client will initialize a connection with the server. To accomplish this, the server must listen to a certain port and the client must know the servers ip and port. In our case, the communications use the TCP and HTTPS protocols. Alpha Zero uses three distinct servers: an AI server, a data server and an Apache web server.

3.1 The Web Server

Alpha Zero’s web server uses the Apache web server application. The web server is used to host static files such as the source code for the iOS client (see section 3.4.2 on page 35), a debug version of the same client, the domain name of the AI server and the domain name of the data server. All these files were located at <https://wandhoven.ddns.net/code/AlphaZero/>

3.2 The Data Server

The data server stores all global information. This was just the elo-rating to begin with, but was later expanded to handle all data. This explains the somewhat strange communication protocol. Requests to the server begin by sending a 4 byte signed integer a identifying the general action the server must perform. The first action $a = 1$ will return the elo-rating of a certain agent. This will require a further 4 bytes identifying the agent. The second action $a = 2$ will set an agents elo-rating. Two 4 byte integers are sent, the first identifying the agent and the second the elo-rating to set to. The third action $a = -1$ will require a 4 byte signed integer e and return the agent’s identifier with an elo-rating equal to r defined as:

$$r = \min(\{x \in \mathbb{E} | x \geq e\}) \quad (54)$$

where \mathbb{E} is the set of the elo-ratings of all agents. The last action $a = -2$ will access the custom data part of the server. This subsection will require

an integer describing how many bytes the request consists of. The request is encoded using the python pickle library and consists of either a tuple containing a string, and a list of strings; or a tuple containing a string, a list of strings, and any other data type. In the first case, the system will return the value of the saved data associated with that request. In the second case, the value of the associated the variable will be set to whatever the third value is.

The first two variables of the tuple are a string f and a list of strings k . f tells the server in which file the variable is stored. Therefore, the server will load the json file with the name f . k is the list of keys used to index the dictionary f .

For example, with $f = \text{example.json}$, the server would decode the json file "example.json" shown in listing 1 on page 32. Assuming that $k = ["\text{address}", "\text{city}"]$, the server would first search for key "address" in the outer most directory. At that key, there is another dictionary, which is then searched for the key "city". At that key, there is a string ("New York"), which would be returned to the client.

```

1  {
2      "firstName": "John",
3      "lastName": "Smith",
4      "isAlive": true,
5      "age": 27,
6      "address": {
7          "streetAddress": "21 2nd Street",
8          "city": "New York",
9          "state": "NY",
10         "postalCode": "10021-3100"
11     },
12     "phoneNumbers": [
13         {
14             "type": "home",
15             "number": "212 555-1234"
16         },
17         {
18             "type": "office",
19             "number": "646 555-4567"
20         }
21     ],
22     "children": [],
23     "spouse": null
24 }

```

Listing 1: example.json
from <https://en.wikipedia.org/wiki/JSON>

3.3 The AI Server

The AI server is used to evaluate a state and determine the best action using Alpha Zero. This is done by sending the server the state and waiting for it to send back the action.

3.3.1 State Transmission

To send a state to the server, the state's 6×7 board is converted into an array of 85 boolean values². The first 42 booleans represent whether or not the starting player has a stone at that position. Positions are mapped from left to right and then top to bottom. The table below shows the order in which the positions will be added to the boolean array. The next 42 booleans are identical to the first but for the non-starting player. The last position tells the server which player is taking the next action.

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41

Consider the following game state (fig 9 on page 33) at which the starting player is at turn.

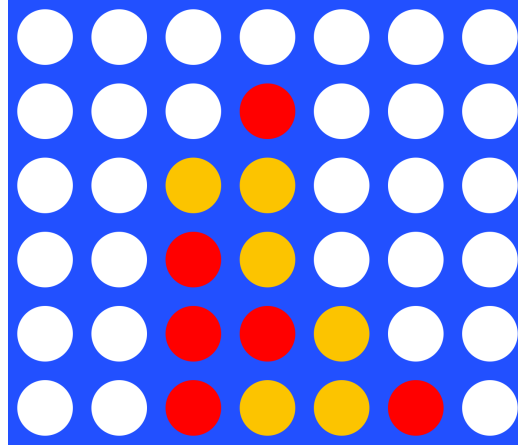


Figure 9: example game state

²Stored as integers 0 and 1 in memory

This state's boolean array a is:

$$a = \text{vec} \left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcolor{red}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{orange}{0} & \textcolor{orange}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{orange}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{red}{1} & \textcolor{orange}{0} & 0 & 0 \\ 0 & 0 & \textcolor{red}{1} & \textcolor{orange}{0} & \textcolor{orange}{0} & \textcolor{red}{1} & 0 \end{bmatrix} \right) \frown \text{vec} \left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \textcolor{red}{0} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{orange}{1} & \textcolor{orange}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{orange}{1} & 0 & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{orange}{0} & \textcolor{orange}{1} & 0 & 0 \\ 0 & 0 & \textcolor{red}{0} & \textcolor{orange}{1} & \textcolor{orange}{1} & \textcolor{red}{0} & 0 \end{bmatrix} \right) \frown \langle 1 \rangle \quad (55)$$

Where vec is the matrix vectorization defined in section 4.5 on page 37 and \frown is the vector concatenation defined in section 4.6 on page 37. To the front of this vector, the version of the AI we want to play against is added. If the version is invalid the server will default to the best version.

3.3.2 Action Selection

The AI's action is selected by running MCTS simulations and choosing the action with the most evaluations. It is the same algorithm as the deterministic method defined in section 1.5.1 on page 24.

3.3.3 Action Transmission

The action is stored and transmitted as a four byte integer.

3.4 Clients

There are two clients for the AlphaZero system. The first is a python client for DOS, macOS, Linux, etc., and the second was developed for iOS using pythonista.

3.4.1 Desktop Client

The Desktop client will allow the player to play against an AI with a slightly better elo-rating than the player's. This is done by creating an account on the data server. The data server will give the client a unique number representing its account. The Client will then proceed to save this number and request the player's elo-rating. Finally, the AI version with the closest but better elo-rating to that of the player is requested. Now the client renders the game board and randomly selects a starting player. The client will then wait for

user inputs and query the server as appropriate, i.e. what player is at turn. When the game is done, the client's log of the game is uploaded to the data server and the player will be shown the appropriate end-of-the-game screen. The client also has the possibility to request game logs and replay games. This client version uses the python socket library for communication and Tkinter to render the game board.

3.4.2 iOS Client

The iOS version does the same thing as the Desktop version with a few differences. Firstly, it renders the board using the pythonista scene library. Secondly, it does not create or store accounts. The player will always play against the best AI version. Thirdly, it does not have the possibility to replay games. Lastly, the Client will request its actual source code from the web server to allow for easier updating.

4 Further definitions

4.1 Set Exclusion

Let $\mathbb{A} - \mathbb{B}$ be the set exclusion of \mathbb{A} and \mathbb{B} .

$$\mathbb{A} - \mathbb{B} = \mathbb{A} \setminus \mathbb{B} = \{x : x \in \mathbb{A} \text{ and } x \notin \mathbb{B}\} \quad (56)$$

4.2 Hadamard product

Let A and B be $2 \times m \times n$ matrices. For all $i \in [1, m]$ and $j \in [1, n]$ the hadamard product $A \circ B$ is defined as:

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij} \quad (57)$$

For example consider the following 2×3 matrices:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & b_{32} \end{bmatrix} \circ \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \\ a_{31}b_{31} & a_{32}b_{32} \end{bmatrix}$$

4.3 Inner Product

4.3.1 Matrices

Let $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{m \times n}$ be two $n \times m$ matrices.

Let their Inner Product $\langle A, B \rangle_I : \mathbb{R}^{m \times n}, \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$ be defined as:

$$\langle A, B \rangle_I = \sum_{i=1}^m \sum_{j=1}^n A_{ij} B_{ij} = \sum_{i=1}^m \sum_{j=1}^n (A \circ B)_{ij} \quad (58)$$

For example consider the following 2×3 matrices:

$$\left\langle \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & b_{32} \end{bmatrix}, \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \right\rangle_I = \begin{aligned} & a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} \\ & + a_{22}b_{22} + a_{31}b_{31} + a_{32}b_{32} \end{aligned}$$

4.3.2 n -dimensional Tensors

Let $A \in \mathbb{R}^{m \times \dots}$ and $B \in \mathbb{R}^{m \times \dots}$ be two n dimensional $m \times \dots$ tensors with $n > 2$

Let the Inner product $\langle A, B \rangle_I : \mathbb{R}^{m \times \dots}, \mathbb{R}^{m \times \dots} \rightarrow \mathbb{R}$ be defined as:

$$\langle A, B \rangle_I = \sum_{i=0}^m \langle A_i, B_i \rangle_I \quad (59)$$

4.4 Submatrix

Let $m = m_{ijk}$ be an $m \times n \times o$ dimensional tensor.

Let $\langle \rangle_{S \times y, ab}$ be the matrix slicing operator. $x \times y$ is the size of the submatrix the operation should output. ab is the top left position of the submatrix within the outer matrix. For the operation to be defined, the following must be true: $x \in [0, m[$, $y \in [0, n[$, $a \in [1, m - x]$ and $b \in [1, n - y]$. The submatrix $\langle m \rangle_{S \times y, ab}$ is defined as:

$$\langle m \rangle_{S \times y, ab} = \begin{bmatrix} [m_{ab1}, \dots, m_{abo}] & \dots & [m_{(a+x)b1}, \dots, m_{(a+x)bo}] \\ \vdots & \ddots & \vdots \\ [m_{a(b+y)1}, \dots, m_{abo}] & \dots & [m_{(a+x)(b+y)o}] \end{bmatrix} \quad (60)$$

4.5 Vectorization

The vectorization of an $m \times n$ matrix A , denoted $\text{vec}(A)$, is the $mn \times 1$ column vector obtained by stacking the columns of the matrix A on top of one another:

$$\text{vec}(A) = [A_{11} \dots A_{1m} A_{21} \dots A_{2m} \dots A_{n1} \dots A_{nm}]^T \quad (61)$$

Taken verbatim from:[\[7\]](#)

For example, the 3×2 matrix $A = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$ vectorizes to

$$\text{vec}(A) = \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \end{pmatrix}$$

4.5.1 Tensors

Let $T \in \mathbb{R}^{n \times \dots}$ be a n -dimensional Tensor. Let $\text{vec}(T)$ be defined as:

$$\text{vec}(T) = T_0 \frown T_1 \frown \dots \frown T_n \quad (62)$$

Where \frown is defined defined in section [4.6](#) on page [37](#). For those who are familiar with python's numpy library, vec is equivalent to `numpy.flatten`.

4.6 Vector Concatination

Vector Concatination of two vectors v and u of dimensions n_v and n_u , denoted $v \frown u$, is the $n_v + n_u$ dimensional vector obtained by placing both vectors one on top of the other.

$$v \frown u = \begin{pmatrix} v_1 \\ \vdots \\ v_{n_v} \\ u_1 \\ \vdots \\ u_{n_u} \end{pmatrix} \quad (63)$$

4.7 Future Assignment Operator

The future assignment operator, denoted \Rightarrow , is in the explanation of iterative algorithms to represent assigning an value to a variable for the next iteration. The operation implies the existence of an index i . Let x represent any variable and v be any possible value of x . On the i^{th} use of the future assignment operator on x .

$$v \Rightarrow x \tag{64}$$

is equivalent to

$$v = x_{i+1} \tag{65}$$

Furthermore any variable referred to, that the uses the future assignment operator, will automatically refer to the last set one. This means that if an algorithm uses x and the initial value of $x = x_0 = z$, where z is a possible value of x . Then that algorithm uses this operation on x , $v \Rightarrow x$. Then from that point forward in the logical operations all references to $x = x_1 = v$. The reason for this is that using indices in the MCTS is more confusing than helpful.

4.8 Sets

Let

$$\mathbb{R}_{a,b} = \{x \in \mathbb{R} : a \leq x \leq b\} \tag{66}$$

References

- [1] Guillaume MJ-B Chaslot, Mark HM Winands, and HJVD Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.
- [2] Arpad E. Elo. *The Rating of Chessplayers, Past and Present*. Arco Pub., New York, 1978.
- [3] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.
- [4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharmashan

- Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.
- [5] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.
- [6] Wikipedia. Vector quantization — Wikipedia, the free encyclopedia. <http://en.wikipedia.org/w/index.php?title=Vector%20quantization&oldid=1082226300>, 2022. [Online; accessed 06-June-2022].
- [7] Wikipedia contributors. Vectorization (mathematics) — Wikipedia, the free encyclopedia. [https://en.wikipedia.org/w/index.php?title=Vectorization_\(mathematics\)&oldid=1057421500](https://en.wikipedia.org/w/index.php?title=Vectorization_(mathematics)&oldid=1057421500), 2021. [Online; accessed 17-March-2022].
- [8] Zhihua Zhang. Artificial neural network. In *Multivariate time series analysis in climate and environmental research*, pages 1–35. Springer, 2018.

Acknowledgements

- Leonie Scheck, Frederik Ott, Nico Steiner, Wolfgang Wandhoven for aiding in evaluating the AI against human players and providing feedback.

5 Examples

5.1 Mcts

The full set of data as seen by the computer in the mcts example (fig 1 on page 11). The node sets are as follows defined by the amount of edged leaving

each node.

$$\mathbb{M} = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\} \quad (67)$$

$$\mathbb{M}_{leaf} = \{n_3, n_5, n_6, n_7, n_8\} \quad (68)$$

$$\mathbb{M}_{expanded} = \{n_1, n_2, n_4\} \quad (69)$$

The edge sets for every node are as follows.

$$\mathbb{E}(n_1) = \{e_1, e_2, e_3\} \quad (70)$$

$$\mathbb{E}(n_2) = \{e_4, e_5\} \quad (71)$$

$$\mathbb{E}(n_3) = \emptyset \quad (72)$$

$$\mathbb{E}(n_4) = \{e_6, e_7, e_8\} \quad (73)$$

$$\mathbb{E}(n_5) = \emptyset \quad (74)$$

$$\mathbb{E}(n_6) = \emptyset \quad (75)$$

$$\mathbb{E}(n_7) = \emptyset \quad (76)$$

$$\mathbb{E}(n_8) = \emptyset \quad (77)$$

The edge by action and node function \mathcal{E} looks af follows:

$$\mathcal{E}(n_1, A_1) = e_1 \quad (78)$$

$$\mathcal{E}(n_1, A_2) = e_2 \quad (79)$$

$$\mathcal{E}(n_1, A_3) = e_3 \quad (80)$$

$$\mathcal{E}(n_2, A_4) = e_4 \quad (81)$$

$$\mathcal{E}(n_2, A_5) = e_5 \quad (82)$$

$$\mathcal{E}(n_4, A_6) = e_6 \quad (83)$$

$$\mathcal{E}(n_4, A_7) = e_7 \quad (84)$$

$$\mathcal{E}(n_4, A_8) = e_8 \quad (85)$$

The node an edge points from function \mathcal{N}_{from} looks af follows:

$$\mathcal{N}_{from}(e_1) = n_1 \quad (86)$$

$$\mathcal{N}_{from}(e_2) = n_1 \quad (87)$$

$$\mathcal{N}_{from}(e_3) = n_1 \quad (88)$$

$$\mathcal{N}_{from}(e_4) = n_2 \quad (89)$$

$$\mathcal{N}_{from}(e_5) = n_2 \quad (90)$$

$$\mathcal{N}_{from}(e_6) = n_4 \quad (91)$$

$$\mathcal{N}_{from}(e_7) = n_4 \quad (92)$$

$$\mathcal{N}_{from}(e_8) = n_4 \quad (93)$$

The node an edge points to function \mathcal{N}_{to} looks af follows:

$$\mathcal{N}_{to}(e_1) = n_3 \quad (94)$$

$$\mathcal{N}_{to}(e_2) = n_2 \quad (95)$$

$$\mathcal{N}_{to}(e_3) = n_4 \quad (96)$$

$$\mathcal{N}_{to}(e_4) = n_5 \quad (97)$$

$$\mathcal{N}_{to}(e_5) = n_6 \quad (98)$$

$$\mathcal{N}_{to}(e_6) = n_6 \quad (99)$$

$$\mathcal{N}_{to}(e_7) = n_7 \quad (100)$$

$$\mathcal{N}_{to}(e_8) = n_8 \quad (101)$$

6 Proofs

6.1 tanh derivative

$\tanh : \mathbb{R} \rightarrow \mathbb{R}$ is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (102)$$

$$\begin{aligned}
\frac{d}{dx} \left[\frac{e^x - e^{-x}}{e^x + e^{-x}} \right] &= \frac{(e^x + e^{-x}) \frac{d}{dx} [e^x - e^{-x}] - (e^x - e^{-x}) \frac{d}{dx} [e^x + e^{-x}]}{(e^x + e^{-x})^2} \\
&= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\
&= \frac{e^{2x} + 2e^x e^{-x} + e^{-2x} - e^{2x} + 2e^x e^{-x} - e^{-x}}{(e^x + e^{-x})^2} \\
&= \frac{4}{(e^x + e^{-x})^2}
\end{aligned} \tag{103}$$

because

$$\operatorname{sech}(x) := \frac{2}{e^x + e^{-x}} \tag{104}$$

using (103) and (104) we can see, that

$$\frac{d}{dx} [\tanh(x)] = \operatorname{sech}(x)^2 \tag{105}$$

6.2 Softmax Derivative

The Softmax $s : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $n \in \mathbb{N}$, is defined as

$$s(v)_i := \frac{e^{v_i}}{\sum_{j \in v} e^j} \tag{106}$$

for all $i \in [1, n]$. Due to the multidimensional nature of this function we are taking the derivative $\frac{d}{dv_j} s(v)_i$, with $i \in [1, n]$ and $j \in [1, n]$.

case $i \neq j$

$$\begin{aligned}
\frac{d}{dv_j} \left[\frac{e^{v_i}}{\sum_{a \in v} e^a} \right] &= - \frac{\frac{d}{dx} [\sum_{a \in v} e^a] e^{v_i}}{(\sum_{a \in v} e^a)^2} \\
&= - \frac{e^{v_j} e^{v_i}}{(\sum_{a \in v} e^a)^2} \\
&= -s(v)_j s(v)_i
\end{aligned} \tag{107}$$

case $i = j$

$$\begin{aligned}
\frac{d}{dv_j} \left[\frac{e^{v_i}}{\sum_{a \in v} e^a} \right] &= \frac{\frac{d}{dx} [e^{v_i}] \sum_{a \in v} e^a - e^{v_i} \frac{d}{dx} [\sum_{a \in v} e^a]}{(\sum_{a \in v} e^a)^2} \\
&= \frac{e^{v_i} \sum_{a \in v} e^a - e^{v_i} e^{v_j}}{(\sum_{a \in v} e^a)^2} \\
&= s(v)_i - s(v)_i s(v)_j \\
&= s(v)_i (1 - s(v)_j)
\end{aligned} \tag{108}$$

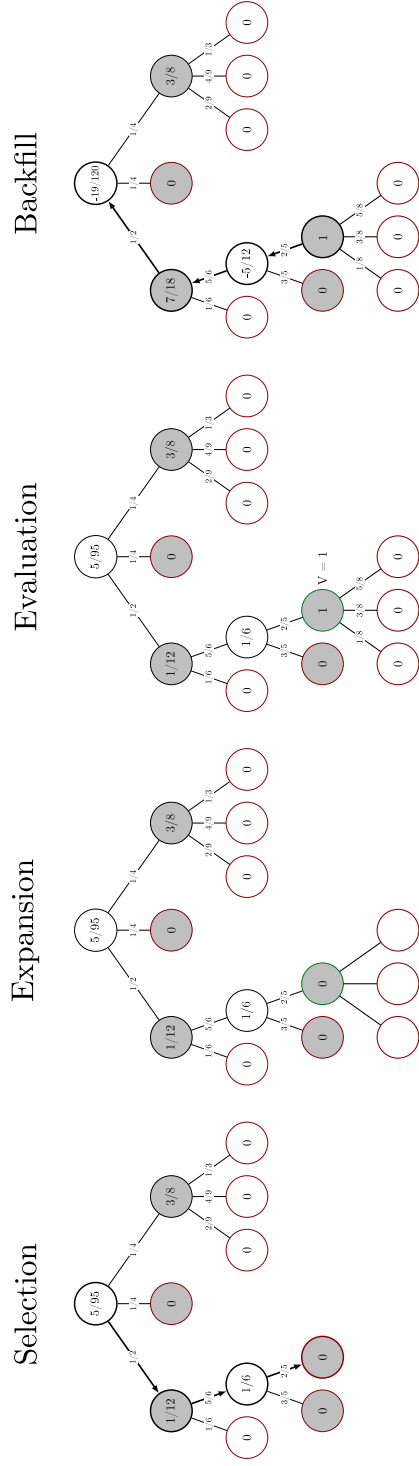


Figure 10: MCTS simulation steps. In this diagram, the numbers in the node represent Q and the number on the arrow is P . The red nodes are leaf nodes and the green one is the leaf node n_L . During the **selection** phase, σ is used to find successive nodes until the leaf node n_L is reached. This is shown with the arrows. During the **expansion** phase, new nodes and edges are added for all possible legal actions at the node n_L . The **evaluation** phase gives the new nodes the following values $Q = 0$ and $P = \pi_a$. The **evaluation** phase gives the new nodes the following values $Q = 0$ and $P = \pi_a$. The value of the leaf v is then used during the **backfill** phase to update the Q 's of all nodes traversed during selection.

Source: modified from <https://en.wikipedia.org/wiki/File:MCTS-steps.svg>
File available under Creative Commons Attribution-Share Alike 4.0 International at <https://wandhoven.ddns.net/edu/AlphaZeroTheory/images/MCTS-steps.svg>