Mastering the game of Connect4 through self-play

Julian Wandhoven
FGZ

August 7, 2023

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 and so forth. This is done by training a neural network from data generated by a Monte Carlo Tree Search. In this project, I modified and implemented the algorithm from scratch and applied it to Connect4. This document also explains the basic mathematics behind neural networks and Monte Carlo Tree Searches. (see articles [**?**][**?**])

Contents

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, an AI will be built and taught to play connect4. The entire algorithm is implemented in C++. Starting from a random computer player (agent), the algorithm is able to improve that agent, and in so doing create a better one. This can be repeated to produce a series of agents, that become progessively better at playing the game. These agents will then be evaluated against each other using the Elo evaluation system [**?**] (see section 2 "Evaluation" on page 26). Additionally, I have added a short introduction on the mathematics behind neural networks in section 1.4 "Neural Network" on page 13.

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 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. Training an AI by teaching it to prefer advantagious actions is called reinforcement learning.

1.1 Reinforcement Learning

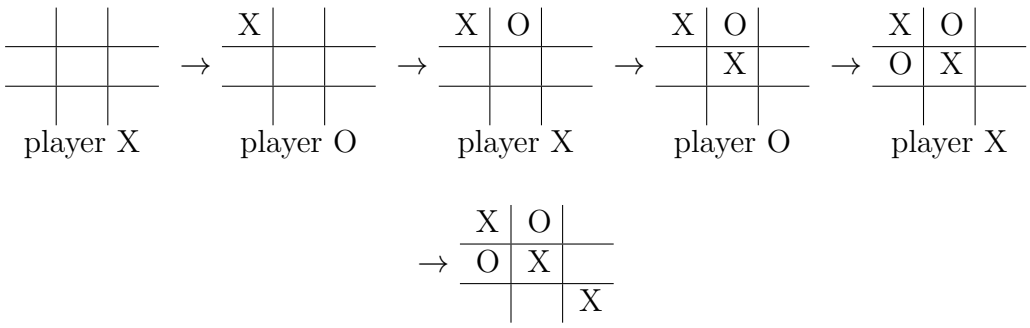
When training neural networks, there are three major possible situations: Supervised learning, unsupervised learning, and reinforcement learning. The first uses predetermined data with known in- and outputs the network is trained to predict. An example of supervised learning is the recognition of handwriting 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 previously 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. Each group is defined by its centroid point and contains all points closer to that centroid point than any other centroid point. Training happens by selecting a random sample point from the input data, and moving the closest centroid point towards this sample point by a fraction of the distance between them.[**?**] This is repeated until an equilibrium is reached. An example of both supervised and unsupervised learning can be found in the demo[1].

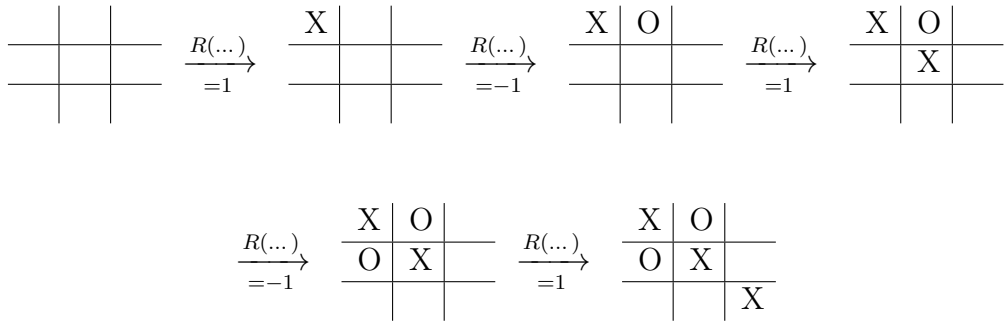
These two methods represent the extreme ends of the spectrum. Reinforcement learning can be thought of as an intermediate form. It uses a

[1] demo is at https://github.com/JulianWww/Matura-AlphaZero-demos

predetermined environment which gives positive, neutral and negative feedback. The neural network is 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 feedback that only occurs many game steps ahead 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 1.5.1.2 "Memory" on page 24). When the game reaches an end state, a winner is determined, and the feedback is propagated backwards up through all states used to get to that end state. 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 for that player, 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 during the game g. Let's look at a tic tac toe example of the following game:



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 positive feedback and discourage actions that lead to negative feedback.

1.2 Game

In order to train a reinforcement learning AI, it must interact with an environment. In Alpha Zero the game is the environment. The game consists of a series 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 over. Every game ends in an end game state. At an end game state, one player won or the game ended in a tie. Together, the successive states form a directed graph. Any possible path through this graph starting from a root state (the inital state of the board) and ending in any end game state, is a possible game. From an end game state, there is nowhere left to go. For connect4 an end game state has either four stones in a line or a full game board. Let $\mathbb{G}$ be the set of all legal game states. Let $\mathbb{G}_{done} \subset \mathbb{G}$ be the set of all game states for which the game is over. At every game state one player is at turn. This means that that player will take an action next. Let $\phi(s) : \mathbb{G} \to \{1, -1\}$ be the function mapping states to players. In the

tick-tack-toe example from earlier we could say that:

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

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 multiple distinct stones. We can represent these game boards as a 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 layers stack.

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

Internally, the game board is represented by a flat vector. The conversion from a game state $s \in \mathbb{G}$ to a vector is defined as $vec(T_s(s))$. Where $T_s(s) : \mathbb{G} \rightarrow \mathbb{R}^{\cdots}$ is the board's 3-dimensional board tensor. The vec function is defined in section 5.4.1 "Tensor Vectorization" on page 34.
This operation for the tic tac toe board from above would look like this:

$$vec\left(T_s\left(\begin{array}{|c|c|c|} \hline & \text{X} & \text{O} \\ \hline \text{O} & \text{X} & \\ \hline \text{O} & & \text{X} \\ \hline \end{array}\right)\right) = [0\,1\,0\,0\,1\,0\,0\,0\,1\,0\,0\,1\,1\,0\,0\,1\,0\,0]^T \tag{2}$$

1.2.2 Actions

Actions are numbers used to identify changes to the game. Every game has a set of all possible actions represented by numbers $\mathbb{A}_{possible} \subset \mathbb{N}_0$. In connect4, the set of all possible actions for the current player is $\mathbb{A}_{possible} = [0, 41] \cap \mathbb{N}$. 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 of actions to game fields is as follows for connect4:

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 $\mathcal{A} : \mathbb{G} \times \mathbb{A} \to \mathbb{G}$ is used to get from one game state to another by taking an action. Where $\mathbb{A}$ is the set of all legal actions for the chosen game state. If we were to map actions to positions for tic tac toe as follows and the current game state were

0	1	2
3	4	5
6	7	8

X	O	

State s

then player X is allowed to place a stone in any empy field $\implies \mathbb{A}(s) = \{2, 3, 4, 5, 6, 7, 8\}$. Therefore, $\mathcal{A}(s, a)$ 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 network's prediction. Because an MCTS changes during simulation, indices are used to specify which simulation step the tree is in. An MCTS simulation consists of three phases: Leaf Selection, Node Evaluation and Expansion as well as Backfill.

An MCTS graph consists of nodes and edges. Nodes represent game states and edges represent actions. As in the game, edges connect nodes to

each other. The amount of nodes and structure of the MCTS is changed by the algorithm itself during simulation. During the first simulation, the tree contains only the root node n_0. The root node is the starting point of the simulation. Let $l \in \mathbb{L}$ be the index of the current simulation step. Let $\mathbb{L} = [0, S] \cap \mathbb{N}$ be the set of all l, where S is a constant defined in the algorithm's configuration file. Let $\mathbb{M}_l \subseteq \mathbb{G}$ be the set of all nodes in the tree at step $l \in \mathbb{L}$. That implies $\mathbb{M}_0 = \{n_0\}$.

Every node $n_l \in \mathbb{M}_l$ has a set of edges $\mathbb{E}_l(n_l)$ at step l that point from it to another node. Let $\mathbb{E}_l$ be the set of all edges in the graph $\mathbb{M}_l$. The function $\mathcal{N}_{to_l} : \mathbb{E}_l \to \mathbb{M}_l$ maps an edge to the node it is pointing to while $\mathcal{N}_{from_l} : \mathbb{E}_l \to \mathbb{M}_l$ 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 leaf nodes. For every node $n_l \in \mathbb{M}_{leaf_l}$, the following is true by definition $\mathbb{E}_l(n_l) = \emptyset$. Expanded nodes have edges leading out of them.

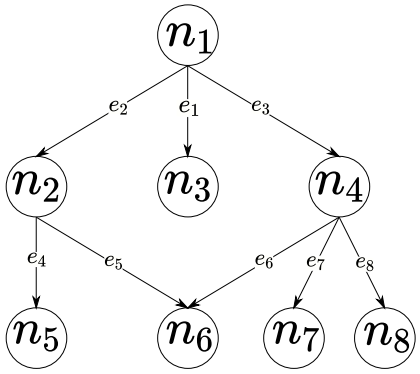


Figure 1: This is a possible MCTS graph. In this example the set of all nodes is $\mathbb{M} = \{n_1, n_2, \ldots, n_8\}$, the set of all leaf nodes is $\mathbb{M}_{leaf} = \{n_3, n_5, n_6, n_7, n_8\}$. The set $\mathbb{E}(n)$ of a node n, beeing the set of edges pointing away from that node $\Longrightarrow$ for node n_1 is $\mathbb{E}(n_1) = \{e_1, e_2, e_3\}$. The same for all other nodes. $\mathcal{N}_{to}(e)$ is the node an edge e points to. This means that $\mathcal{N}_{to}(e_1) = n_3$, $\mathcal{N}_{to}(e_2) = n_2$, etc. In contrast $\mathcal{N}_{from}(e)$ is the node an edge e points from.
Therefore, $\mathcal{N}_{from}(e_1) = n_1$, $\mathcal{N}_{from}(e_2) = n_1$, etc.

1.3.1 Evaluation Basis

The MCTS's goal is to find a good estimation of the reward for a certain action at a certain state. This reward estimation is $Q_l : \mathbb{E}_l \to \mathbb{R}$. To define Q_l, the functions $W_l : \mathbb{E}_l \to \mathbb{R}$ and $N_l : \mathbb{E}_l \to \mathbb{N}_0$ are required. $N_l(e_l)$ is the amount of times an edge $e_l \in \mathbb{E}_l$ has been traversed during simulation. $W_l(e_l)$ is the sum of the reward computations from all $N_l(e_l)$ times the edge has been evaluated. With these two functions Q_l is defined as:

$$Q_l(e_l) = \begin{cases} 0 & N_l(e_l) = 0 \\ \frac{W_l(e_l)}{N_l(e_l)} \end{cases} \tag{3}$$

In a classic MCTS this is all that would be required. However, Alpha Zero uses an improved MCTS that increases computational efficency by exploring more promissing paths more early on. In order to accomplish this, the neural network gives an estimation of the advantagiousness of a certain action at a certain state. This estimation is the edges policy $P_l : \mathbb{E}_l \to \mathbb{R}$. The policy approximates

$$\frac{N_{mean}(e_l)}{\sum_{e \in \mathbb{E}_l(\mathcal{N}_{from}(e_l))} N_{mean}(e)} \tag{4}$$

where N_{mean} is the average N_S over all previous simulations.

1.3.2 Leaf Selection

MCTS's evaluation starts by simulating future moves within the tree starting from n_0. 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 $n_l \in \mathbb{M}_l$ the selection function $\sigma_l : \mathbb{M}_l \to \mathbb{E}_l$ is used. To define σ_l we must first define the edge evaluation function $u_l : \mathbb{E}_l \to \mathbb{R}$. u_l is defined as:

$$u_l(e) = Q_l(e) + c \cdot P_l(e) \cdot \frac{\sqrt{\Sigma_{b \in \mathbb{E}(\mathcal{N}_{from_l}(e))} N_l(b)}}{1 + N_l(e)} \tag{5}$$

where $c \in \mathbb{R}^+$ is the exploration constant used to define how important exploration is. The smaller c is, the more important is Q and the less important

are exploration and P. For my configuration $c = 2$ turned out to work fine. σ_l, for a given node $n_l \in \mathbb{M}_l$, is then defined as:

$$\sigma_l(n_l) = \text{argmax}_l(\mathbb{E}_l(n_l)) \tag{6}$$

argmax_l returns the edge $e \in \mathbb{E}_l$ with the largest $u_l(e)$.
In order to find a leaf node n_{L_l} starting from the root node n_{0_l} and to be able to update the tree later on, the following algorithm is run until a leaf node is found. First, let $i \in \mathbb{N}_0$ be the index of the iteration of this algorithm. Let $\mathbb{E}_{back,l,i} \subseteq \mathbb{E}_l$ be the set of edges traversed by the algorithm at step i during simulation l. By definition $\mathbb{E}_{back,l,0} := \emptyset$. Then:

$$n_{(i+1)_l} = \mathcal{N}_{to}(\sigma_l(n_{i_l})) \tag{7}$$

$$\mathbb{E}_{back,l,i+1} = \mathbb{E}_{back,l,i} \cup \{\sigma_l(n_{i_l})\} \tag{8}$$

Once a leaf-node $n_{L_l} \in \mathbb{M}_{leaf_l}$ is reached at step $i = L$ that node is evaluated. $\mathbb{E}_{back,l,L}$ will be important in section 1.3.4 "Backfill" on page 12.

1.3.3 Node Evaluation and Expansion

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

case: end game state $n_{L_l} \in \mathbb{G}_{done}$

In this case, the tree is not changed but backfill is performed with the reward value $v_l \in \{0, \pm 1\}$:

$$v_l = -R(n_{(L-1)_l}, e) \tag{9}$$

where e is the edge taken at node $n_{(L-1)_l}$ to reach n_{L_l}.
(The value of v_l is "negative" because it is player B's turn after player A won.)

case: not end game state $n_{L_l} \notin \mathbb{G}_{done}$

In this case, the node is passed to the neural network (section 1.4 "Neural Network" on page 13). The neural network evaluates the node and outputs a policy vector $\pi_l \in [0, 1]^{|\mathbb{A}|}$ and an estimated reward scalar $v_l \in [-1, 1]$. 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 pointing 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 the edges in $\mathbb{E}_{new}$ are pointing to. The various tree sets and functions are then updated as follows for step $l+1$.

$$\mathbb{M}_{l+1} = \mathbb{M}_l \cup \mathbb{M}_{new} \tag{10}$$

$$\mathbb{E}_{l_+1} = \mathbb{E}_l \cup \mathbb{E}_{new} \tag{11}$$

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

$$\mathbb{M}_{leaf_{l+1}} = \{n \in \mathbb{M}_{l+1} \ : \ \mathbb{E}_{l+1}(n) = \emptyset\} \tag{13}$$

$$P_{l+1}(e) = \begin{cases} P_l(e) & e \in \mathbb{E}_l \\ \pi_{l_e} \end{cases} \tag{14}$$

1.3.4 Backfill

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

$$N_{l+1}(e) = \begin{cases} 0 & e \in \mathbb{E}_{new} \\ N_l(e) + 1 & e \in \mathbb{E}_{back,l,L} \\ N_l(e) \end{cases} \tag{15}$$

$$W_{l+1}(e) = \begin{cases} 0 & e \notin \mathbb{E}_l \\ W_l(e) + v_l \cdot \phi(n_{L_l})\phi(\mathcal{N}_{from}(e)) & e \in \mathbb{E}_{back,l,L} \\ W_l(e) \end{cases} \tag{16}$$

The "ϕ multiplication" $\phi(n_{L_l})\phi(\mathcal{N}_{from}(e)) = \pm 1$ is needed to reward the moves of the winning player and to punish the moves of the other player. Q_{l+1} is already defined according to (3). In contrast to [**?**], this implementation stores the MCTS simulation values in the edges to allow for the fact that actions from two different 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 accordingly.

1.4 Neural Network

Search algorithms like MCTS are able to find advantageous action sequences. Traditionally, like in "stockfish 8" for example, 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 instead a biheaded deep convolutional neural network trained by information gathered from MCTS simulations. 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 [**?**]. There are many different feed-forward neural network layers and activation functions to chose 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 $1 \times n$ dimentional 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 $\delta_{wb} : \mathbb{R}^n \to \mathbb{R}^m$

$$\delta_{wb}(x) = w \cdot x + b \tag{17}$$

1.4.1.2 Convolutional Layer

Convolutional layers are commonly used for image processing. They are used here to perform image processing of the "image" of the game board. Their input is a 3d-tensor $I \in \mathbb{R}^{i \times j \times l}$. 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 $m \times n$ is defined for the layer. Every kernel $k \in \mathbb{K}$ consists of a weight tensor $w_k \in \mathbb{R}^{m \times n \times l}$ and an optional bias scalar $b_k \in \mathbb{R}$. These kernels are then placed over every possible location of the input tensor I, and an output is computed for that position and kernel. This is useful to recognize patterns, that are independant of location. For every kernel $k \in \mathbb{K}$, the kernel's forward operation $\xi_k : \mathbb{R}^{m \times n \times l} \to \mathbb{R}$ is defined as:

$$\xi_k(I_{sub}) = w_k \diamond I_{sub} + b_k \tag{18}$$

where $\diamond$ is the Tensor inner product defined in equation 63 on page 33 and I_{sub} is a subtensor of I. The convolutional operation $\Lambda : \mathbb{R}^{i \times j \times l} \to \mathbb{R}^{i-m+1 \times j-n+1 \times |\mathbb{K}|}$ is an element wise opperation. Given that $I \in \mathbb{R}^{i \times j \times l}$ is the layer input, every element of $\Lambda(I)_{abc}$ with $a \in [1, i-m+1] \cap \mathbb{N}$, $b \in [1, j-n+1] \cap \mathbb{N}$ and $c \in [1, |\mathbb{K}|] \cap \mathbb{N}$ is defined as:

$$\Lambda(I)_{abc} = \xi_{k_c}(\langle I \rangle_{m \times n,\, ab}) \tag{19}$$

where k_c is the c^{th} kernel and $\langle\rangle$ is the submatrix operation as defined in section 5.3 "Submatrix" on page 34.

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 $\Lambda(I) \in R^{2\times 2\times 1}$. To calculate $\Lambda(I)_{111}$, we compute the kernel operation $\xi_{k_1}(\langle I \rangle_{S_{3\times 3,\,11}})$

$$\Lambda\left(\begin{bmatrix}[3] & [0] & [1] & [5]\\ [2] & [6] & [2] & [4]\\ [2] & [4] & [1] & [0]\\ [3] & [0] & [1] & [5]\end{bmatrix}\right)_{111} = \begin{bmatrix}[3] & [0] & [1]\\ [2] & [6] & [2]\\ [2] & [4] & [1]\end{bmatrix} \diamond \begin{bmatrix}[-1] & [0] & [1]\\ [-2] & [0] & [2]\\ [-1] & [0] & [1]\end{bmatrix} + 7 \quad (20)$$

$$= -1 \cdot 3 + 0 \cdot 0 + 1 \cdot 1 - 2 \cdot 2 + 0 \cdot 6 + 2 \cdot 2 - 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 1 + 7$$
$$= 4$$

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 $f_1(x)$ and $f_2(x)$, the chained function $f(x) = f_1(f_2(x))$ is also linear. 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:

$\mathbb{R} \to \mathbb{R}$

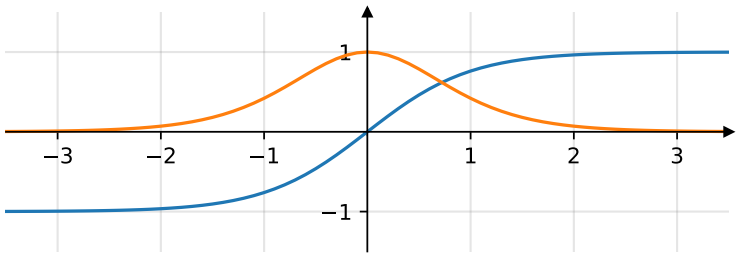


Figure 2: tanh function in blue and the tanh's derivative in orange

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{21}$$

$$\frac{d}{dx} tanh(x) = sech(x)^2 \tag{22}$$

Therefore, for a given vector $v \in \mathbb{R}^n$, $tanh(v)$ shall be defined as:

$$tanh(v) = \begin{pmatrix} tanh(v_1) \\ tanh(v_2) \\ \vdots \\ tanh(v_n) \end{pmatrix} \tag{23}$$

For simple use later on, the derivative of $tanh(v)$, $tanh'(v) \in \mathbb{R}^{n \times n}$ shall be a $n \times n$ dimensional matrix, defined as

$$tanh'(v)_{ij} = \begin{cases} tanh'(v_i) & i = j \\ 0 \end{cases} \tag{24}$$

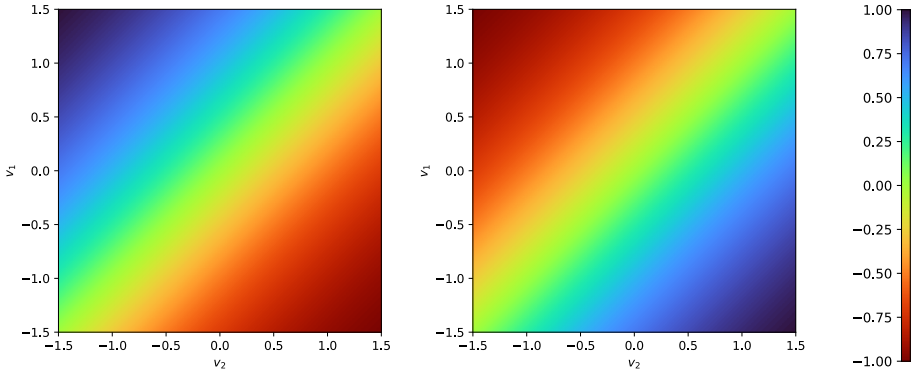
for every element i, j in the derivative matrix.

softmax:

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

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

$$o_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j \in v} e^j} \tag{25}$$



(a) o_1 (b) o_2 scale

Figure 3: Graph of the softmax function from $\mathbb{R}^2 \to \mathbb{R}^2$.

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

case $j = i$:

$$\frac{d}{dv_j}\left[\frac{e^{v_i}}{\sum_{b \in v} e^b}\right] = softmax(v)_i \cdot (1 - softmax(v)_j) \tag{26}$$

case $j \neq i$:

$$\frac{d}{dv_j}\left[\frac{e^{v_i}}{\sum_{b \in v} e^b}\right] = -softmax(v)_i \cdot softmax(v)_j \tag{27}$$

Therefore, the derivative of the softmax function is:

$$softmax'(v)_{ij} = \begin{cases} softmax(v)_i \cdot (1 - softmax(v)_j) & i = j \\ -softmax(v)_i \cdot softmax(v)_j & i \neq j \end{cases} \tag{28}$$

LeakyReLU:

$\mathbb{R} \to \mathbb{R}$

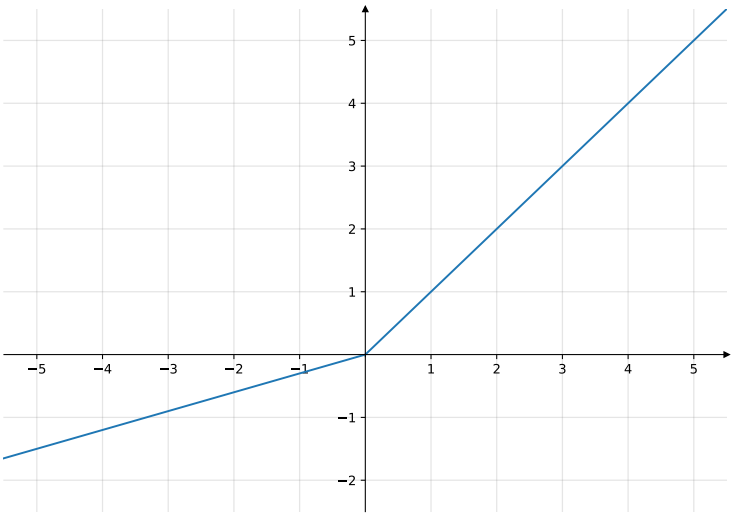


Figure 4: LeakyReLU with $c = 0.3$

$$\text{LeakyReLU}(x) = \begin{cases} x & x \geq 0 \\ x \cdot c & x < 0 \end{cases} \tag{29}$$

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & x > 0 \\ c & x < 0 \end{cases} \tag{30}$$

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 befined for all $x \in \mathbb{R}$. A possible definition that accomplishes the objective is:

$$g(x) = \begin{cases} 1 & x \geq 0 \\ c & x < 0 \end{cases} \tag{31}$$

Therefore, for a given vector $v \in \mathbb{R}^n$, LeakyReLU(v) shall be defined as:

$$\text{LeakyReLU}(v) = \begin{pmatrix} \text{LeakyReLU}(v_1) \\ \text{LeakyReLU}(v_2) \\ \vdots \\ \text{LeakyReLU}(v_n) \end{pmatrix} \tag{32}$$

For simple use later on, the derivative shall be a $n \times n$ dimensional matrix, defined as

$$\text{LeakyReLU}'(v)_{ij} = \begin{cases} g(v_j) & i = j \\ 0 \end{cases} \tag{33}$$

for every element i, j in the derivative matrix.

1.4.1.4 Training introduction

To make this explanation easier, I will use a fully connected neural network and assume we are using supervised learning. Neural network training or backpropagation can be mathematically expressed as minimizing a loss function $\ell(Y_{pred}, Y_{true})$ 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 ($mse = \ell$) loss function is used. $\ell : \mathbb{R}^\mu \times \mathbb{R}^\mu \to \mathbb{R}$ is defined as:

$$\ell(Y_{pred}, Y_{true}) = \frac{|Y_{pred} - Y_{true}|^2}{\mu} \tag{34}$$

The network then performs gradient descent to find parameters that minimize ℓ. Let $\ell' : \mathbb{R}^\mu \times \mathbb{R}^\mu \to \mathbb{R}^\mu$ be the derivative of ℓ.

$$\ell'(Y_{pred}, Y_{true}) = \frac{2}{\mu}\left(Y_{pred} - Y_{true}\right) \tag{35}$$

Given a neural network consisting of n layers, training will start at the last layer and propagate throught the network from back to front. Hence the name backpropagation. To improve the parameters w, b (weight and bias) of every layer in the network, the algorithm must determine in which direction and by how much the variables should be moved. Let $j \in \mathbb{N}_0 \cap [0, n[$ be the index of the layer. The indices start at the first layer ($j = 0$) and count

upwards. In general, every layer is followed by an activation function. The layer and activation function at any particular index j shall be referred to as $f_{a_j}(f_j)$. For the last activation function $j = n - 1$, the change to its output ΔY_j is described as follows using (35):

$$\Delta Y_j = \ell'(Y_{pred}, Y_{true}) \tag{36}$$

In general, f_j is always followed by an activation function f_{a_j}. Let A_j be the precomputed inputs to f_{a_j}. Let ΔA_j be the desired for change to A_j.

$$\Delta A_j = f'_{a_j}(A_j)\Delta Y_j \tag{37}$$

Next comes the update to the weight matrix w_j. Let Δw_j describe the change to w_j and let X_j be the input vector of the layer. Δw_j is then defined as:

$$\Delta w_j = \Delta A_j \cdot X_j^T \tag{38}$$

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

$$\Delta b_j = \Delta A_j \tag{39}$$

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

$$\Delta Y_{j-1} = \Delta A_j \cdot w_j^T \tag{40}$$

This process is repeated until the foremost layer of the neural network is reached. This layer has the index $j = 0$. At the end, the changes are scaled by the learning rate l_r and added to their appropriate values.

1.4.2 Network used by Alpha Zero

The neural network in Alpha Zero is used to estimate the value v and policy p for any node n. v is the neural network's estimation of the state's expected reward. The policy $\pi \in \mathbb{R}^{|\mathbb{A}|}$ 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's 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 $i_n = [X, Y] \in \mathbb{R}^{7\times6\times2}$. Consider the following Connect4 board (fig 5 on page 21).

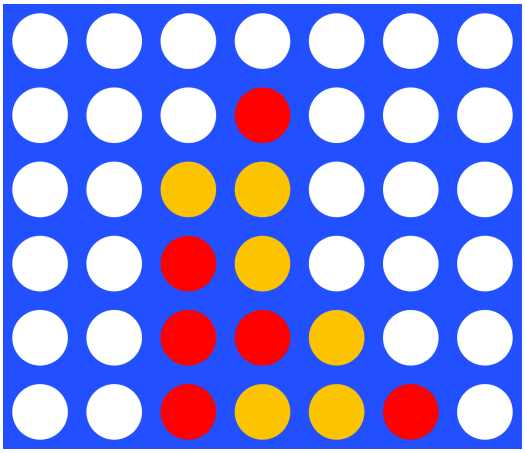


Figure 5

If red is the current player then:

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

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

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

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

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.
In my code, I chose the convolutional block to consist of the following:

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

For every residual block, I chose the following modules:

1. A convolutional layer consisting of 75 filters with a kernel size of 3 x 3
2. Batch normalization
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
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. I used a value head consisting 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

My 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 42
3. A LeakyReLU activation function

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 throught the softmax function to form π:

$$\pi = \text{softmax}(p_{masked}) \tag{41}$$

1.4.2.3 Model Training

Training is performed in batches of 256 states. Both heads are updated using *mse* (Mean-squared-error). In the policy network, all non-legal actions are ignored. This avoids unnecessary updating of the neural network. The value of the value head, 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 to train for, for a given legal action $a \in \mathbb{A}(n)$ is:

$$p_a = \frac{N(n,a)}{\sum_{b \in \mathbb{A}(n)} N(n,b)} \tag{42}$$

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

$$p_a = p_{pre_a} \tag{43}$$

˘

1.5 Training loop

The Alpha zero training loop consists of data generation, training and testing. During data generation, data is generated, that is then used to train the neural network. After every training session, the new model is evaluated against the old one, to determine if it is better. If this is the case, the new model is used to generate data, if not, the old one generates data again and may be replaced after the next round.

1.5.1 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.

1.5.1.1 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 $a = argmax(N(a)), a \in \mathbb{A}(n_t)$ 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:

$$\mathcal{P}(X = a_i, n_t) = \frac{N(a_i)}{\sum_{j \in \mathbb{A}(n_t)} N(j)} \tag{44}$$

($\mathbb{A}(n_t)$ are the allowed actions for state n_t.) 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's data generation phase, actions are always selected deterministically.

1.5.1.2 Memory

The memory is used to store game states, after they where generated, but before they are used to train the neural network. It 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 $r \in \{\pm 1, 0\} = R(g, a)$ 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:

$$v_a = \begin{cases} \mathcal{P}(X = a, \mathcal{N}(g)) & a \in \mathbb{A}(g) \\ p_{pre_a} \end{cases} \tag{45}$$

$\mathbb{A}(g)$ is the set of all legal actions. p_{pre} is defined in section 1.4.2.2 "Neural Network Architecture" on page 22, and is used for all non legal actions. $\mathcal{P}$ is defined in equation 44 on page 24.

1.5.1.3 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 Model Training

Once the memory size exceeds 30'000 states, the self-playing stops. Batches of 256 states are then randomly taken from all the states in the memory and used to train the neural network as described in section 1.4.2.3 "Model Training" on page 23.

1.5.3 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. [**?**]

2.1 Elo-rating

The elo-rating system assigns a number $r_p \in \mathbb{R}$ to every player p. 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 [**?**]:

$$E = \frac{1}{1 + e^{(r_b - r_a)/400}} \tag{46}$$

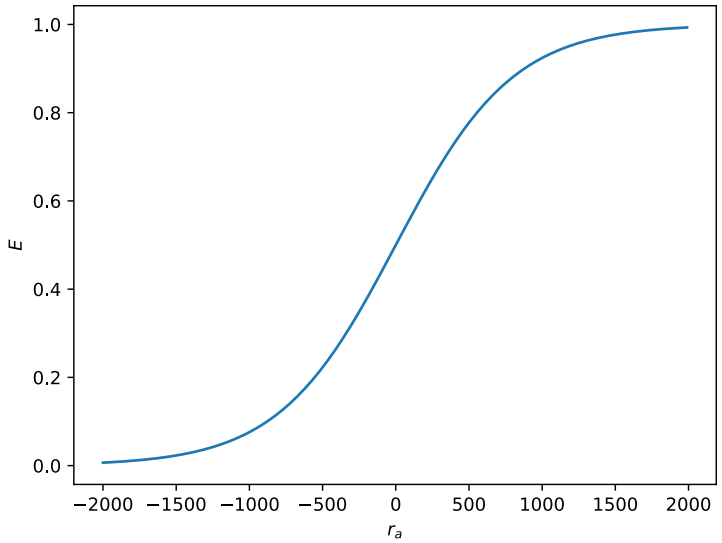


Figure 6: elo-rating win probability for $r_b = 0$

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, a's elo rating is updated as follows[**?**]:

$$r_{a_{n+1}} = r_{a_n} + K(W - E) \tag{47}$$

with:

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

r_{a_n} the current rating of the player.

$W = s_a$ which is defined by equation 48 where a is the player to be updated.

E the expected chance of winning, see equation 46.

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 [**?**]:

$$s_a = \frac{1}{m} \sum \begin{cases} 1 & a \text{ wins} \\ \frac{1}{2} & \text{tie} \\ 0 & a \text{ looses} \end{cases} \tag{48}$$

Assuming that this is the probability of a winning against b, a's elo-rating can be computed by solving equation 46 to r_a (fig 7 on page 28):

$$r_a = r_b - ln\left(\frac{1 - s_a}{s_a}\right) \cdot 400 \tag{49}$$

Since a ranking of all the agents already exists (see section 1.5.3 "Model evaluation" on page 25), an agent's elo-rating can be computed by playing against an older version and then using equation 49 to determine its elo-rating.

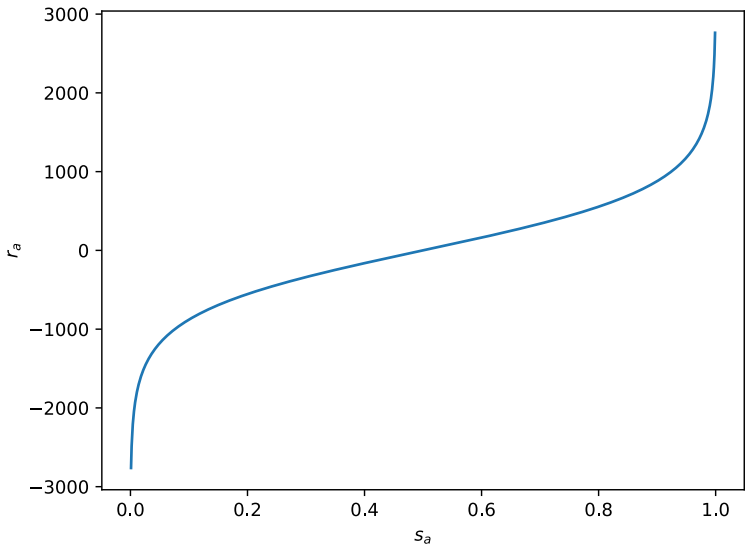


Figure 7: elo inverse function for $r_b = 0$

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 I could find, 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:

$$\Delta r_{min} = -ln\left(\frac{1 - \Delta s_i}{\Delta s_i}\right) \cdot 400 \tag{50}$$

Where Δs_i is the minimal expected chance of agent i beating agent $i-1$. 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_i and s_{i-1} are the scores of agents i and $i-1$, that play aginst each other, then by definition:

$$s_i + s_{i-1} = 1 \tag{51}$$

Let w_i be the win count of agent i over game count.
Let d_i be the draw count over game count.
Let l_i be the loose count of agent i over game count.
Then:

$$s_i = w_i + \frac{d_i}{2} \tag{52}$$

$$s_{i-1} = l_i + \frac{d_i}{2} \tag{53}$$

Then due to θ:

$$w_i \geqslant l_i\theta \tag{54}$$

And by extension:

$$s_i \geqslant s_{i-1} \tag{55}$$

Under the assumption, that there are no ties:

$$s_i \geqslant s_{i-1} \cdot \theta \tag{56}$$

Let s_{min_i} be the minimal possible s_i (assuming no ties). Using (56) and (51):

$$\left| \begin{aligned} s_i + s_{i-1} &= 1 \\ s_i &= s_{i-1}\theta \end{aligned} \right| \tag{57}$$

$$\implies s_i = \frac{\theta}{1+\theta} \tag{58}$$

For $\theta = 1.3$ this means that the expected average change in rating Δr using (58) and (50):

$$\Delta r \geqslant -ln\left(\frac{1}{\theta}\right) \cdot 400 = \Delta r_{min} \cong 105 \tag{59}$$

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

$$\theta = \frac{1}{e^{\frac{-\Delta r}{400}}} \cong 2.8 \tag{60}$$

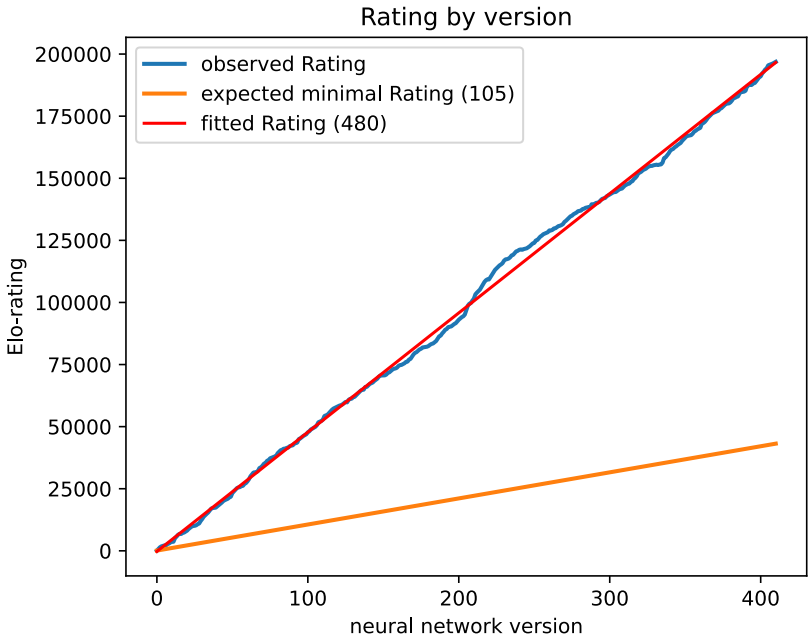


Figure 8: Elo-rating of agents based on their version along with the expected minimal rating Δr_{min} and the best fitted rating Δr.

3 Implementation

I began by implementing the MCTS and neural network seperately. First, I tried using Tensorflow for the neural network evaluation, but its C++ API is nearly undocumented and very complicated to use. So, I settled for Pytorch. With that solved, I implemented the basic training loop and ran the algorithm. This is when the problem of insufficent computational resources occurred. In an attempt to solve this, I ran the program over google cloud platform and moved model evaluation to the GPU. Furthermore, I made game simulation and data generation asynchronous to avoid CPU/GPU downtime while the other is computing. Additionally, I created two python GUI's and one C++ UI that comunicate with a C++ AI server to allow humans to play against the AI. Because the AI server was turned off most of the time but I still wanted to access the elo data and game logs, I created a dedicated data server to store this information. Then, because the AI server running on google cloud platform changes IP a lot, I created a webpage on a webserver, to redirect the clients to the right server. When I aquired a computer capable of handling the AI, a lot of this work became redundant. I also implemented a neural network in numpy and created supervised and unsupervised learning examples.

4 Conclusion

I very much enjoyed the development of the AI as well as deriving the mathematics behind neural networks. Coding was a lot of fun though the endless debugging was painful. Performance issues were a major part of my concern. Solving these was an interesting experience. Working on asynchronous programs of this scale was a new experience for me, and it worked quite well. As a result, I successfully developed an AI-based computer program that plays connect4, and has beaten every human so far that it played against. I am very happy with the results. However, if I were ever to do this again, I would stick to python.
Some improvements could be made - like the following

4.1 Improvements

- The algorithm could be completely rewritten to allow for training on TPU's (tensor processing units).
- Process distribution could be improved to allow for training of the AI on multiple GPU's.
- The algorithm could be tested for different games and other configurations.
- Since most of the time is spent on running neural network simulations, process efficiency could be improved by not only updating MCTS graphs to the current node, but all the way back to the game's root. Furthermore, not only updating the path but every possible path back would increase efficency even further - at least in theory.

5 Appendix: Further definitions

5.1 Hadamard product

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

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij} \tag{61}$$

For example consider the following 2×3 matrices:

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

5.2 Inner Product

5.2.1 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 $A \diamond B \ : \ \mathbb{R}^{m \times n}, \ \mathbb{R}^{m \times n} \to \mathbb{R}$ be defined as:

$$A \diamond B = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} B_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} (A \circ B)_{ij} \tag{62}$$

For example consider the following 2×3 matrices:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & b_{32} \end{bmatrix} \diamond \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{array}{l} a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} \\ + \ a_{22}b_{22} + a_{31}b_{31} + a_{32}b_{32} \end{array}$$

5.2.2 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 $A \diamond B : \mathbb{R}^{m \times \dots}, \mathbb{R}^{m \times \dots} \to \mathbb{R}$ be defined as:

$$A \diamond B = \sum_{i=0}^{m} A_i \diamond B_i \tag{63}$$

5.3 Submatrix

Let $M \in \mathbb{R}^{m \times n \times o}$ be a tensor of size $m \times n \times o$.
Let $\langle\rangle_{x \times y,\, ab}$ 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: $x \in [1, m[\cap\mathbb{N}$, $y \in [1, n[\cap\mathbb{N}$, $a \in [1, m - x + 1] \cap \mathbb{N}$ and $b \in [1, n - y + 1] \cap \mathbb{N}$. The submatrix $\langle M \rangle_{x \times y,\, ab}$ is defined as:

$$\langle M \rangle_{x \times y,\, ab} = \begin{bmatrix} [M_{ab1}, \ldots, M_{abo}] & \ldots & [M_{(a+x-1)b1}, \ldots, M_{(a+x-1)bo}] \\ \vdots & \ddots & \vdots \\ [M_{a(b+y-1)1}, \ldots, M_{a(b+y-1)o}] & \ldots & [M_{(a+x-1)(b+y-1)1}, \ldots, M_{(a+x-1)(b+y-1)o}] \end{bmatrix} \tag{64}$$

5.4 Vectorization

"[…] the vectorization of an $m \times n$ matrix A, denoted vec(A), is the $mn \times 1$ column vector obtained by stacking the columns of the matrix A on top of one another"[**?**]:

$$\text{vec}(A) = [A_{11} \ \ldots \ A_{1m} \, A_{21} \ \ldots \ A2m \ \ldots \ A_{n1} \ \ldots \ Anm]^T \tag{65}$$

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

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

5.4.1 Tensor Vectorization

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

$$vec(T) = vec(T_0) \frown vec(T_1) \frown \cdots \frown vec(T_n) \tag{66}$$

Where $\frown$ is defined defined in section 5.5 "Vector Concatination" on page 35. For those familiar with python's numpy library, vec is equivalent to numpy.flatten.

5.5 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.

$$v \frown u = \begin{pmatrix} v_1 \\ \vdots \\ v_{n_v} \\ u_1 \\ \vdots \\ u_{n_u} \end{pmatrix} \tag{67}$$

5.6 Source Code Locations

The source code can be found in the following github repositories:

- AI, AI-Server, data server and UI's:
  https://github.com/JulianWww/AlphaZero
- Supervised and unsupervised learning demos:
  https://github.com/JulianWww/Matura-AlphaZero-demos
- Neural network in python: https://github.com/JulianWww/NeuralNetwork
- This Document: https://github.com/JulianWww/MaturArbeitAlphaZero

Acknowledgements

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