Mastering the game of Connect 4 through self-play

Julian Wandhoven
fgz

June 13, 2022

Abstract

Alpha Zero is an AI algorythem, 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 Serch. This document also explains how neural networks work and a short explenation of the infrastructure around the AI to allow for playing on remote devices. [?][?]

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 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. 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 27) for all agents. This server only handles routing and data storage while the other one handles the AI. Additionally, I have added a short introduction on how neural networks work and how they are trained in section 1.4 on page 15.

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.

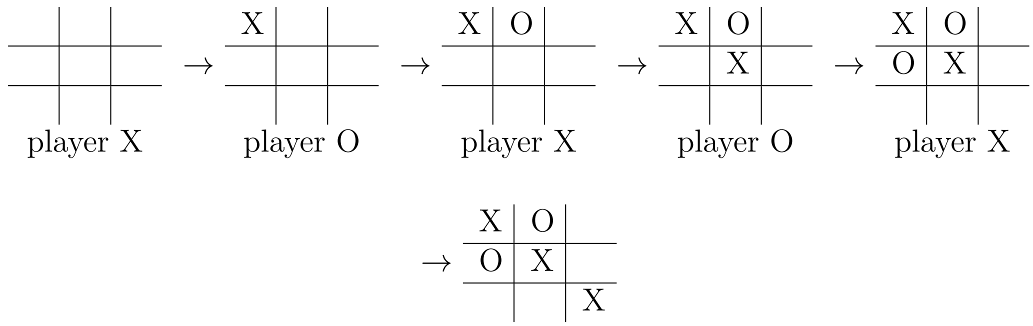
1.1 Reinforcement Learning

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.

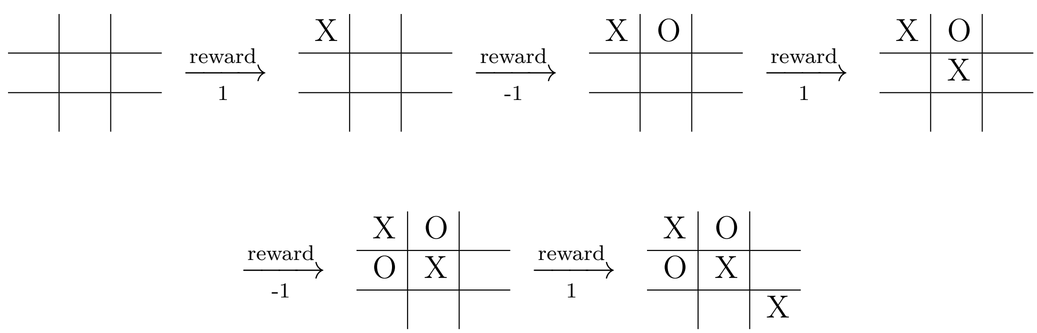
Unsupervised learning or self-organization is used when there is no previous available data and the neural network has to create those classifications itself. An example of unsupervised learning is Vector quantization. The algorythems sorts points in n-dimensional space into a predetermined amount of 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 betwean them. The sample point is selected from the input data [**?**]. An example of both supervised and unsupervised learning can be seen in the demo[1].

These two methods represent the extreme ends of the spectrum. Reinforcement leaning 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 agents 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 26). 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 $R(s, a_s) = 0$. Let g be the set of states traversed in anny given game. Every agent p will try to maximize $\sum_{s \in g \cap \mathbb{P}} R(s, a_s)$. Where $\mathbb{P}$ is the set of all state player p can traverse. Let's look at a tic tac toe example of the following game:

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



Since player X won the game, the reward for every state $s \in g \cap \mathbb{X}$ is $R(s, a_s) = 1$ and the reward for every state $s \in g \cap \mathbb{O}$ is $R(s, a_s) = -1$. Where $\mathbb{X}$ is the set of all state with player X at turn and $\mathbb{O}$ for player O. 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.

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.

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

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 $T_s(s) : \mathbb{G} \rightarrow \mathbb{R}^{\cdot\cdot\cdot}$ is the board's 3-dimensional board tensor. The vec function is defined in section 4.5.1 on page 38. This operation for the tic tac toe board from before would look like this:

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

1.2.2 Actions

Actions are numbers used to identify changes to the game. Every game has a set of all possible actions $\mathbb{A}_{possible} \subset \mathbb{N}_0$. This set represents the sum of actions a certain player can take at anny valid game state. In connect4 there is no need to distinguish betwean colors as the action represents where to place the next stone. The color is determined by which player is at turn when the action is taken. 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 $\mathcal{A} : \mathbb{G} \times \mathbb{A} \to \mathbb{G}$ is used to get from one game state to another by taking an action. Where $\mathbb{A}$ is the set of all legal actions 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
3	4	5
6	7	8

X	O	

State s

In this example player X is allowed to place a stone in anny empy field $\mathbb{A}(s) = \{2, 3, 4, 5, 6, 7, 8\}$. Therefor $\mathcal{A}(s, a)$ is valid if $a \in \mathbb{A}(s)$ and otherwise invalid. Therefor $\mathcal{A}(s, 0)$ and $\mathcal{A}(s, 1)$ are invalid while $\mathcal{A}(s, 2)$, ..., $\mathcal{A}(s, 8)$ are valid.

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.

1.3.1 Structure

This graph consists of nodes that are used to simulate possible sequences of states starting from a root state and edges that connect nodes. The set of all possible nodes $\mathbb{M}_{possible} = \{\mathcal{N}(s) \mid s \in \mathbb{G}\}$, where $\mathcal{N} : \mathbb{G} \to \mathbb{M}_{possible}$ is a bijective function that maps a game state to a node. $\mathcal{S} : \mathbb{M}_{possible} \to \mathbb{G} = \mathcal{N}^{-1}$ is the inverse of $\mathcal{N}$. The set of all allowed actions for a certain node $n \in \mathbb{M}_{possible}$ is $\mathbb{A}(n) = \mathbb{A}(\mathcal{S}(n))$. The set of all nodes in an MCTS at any given time is $\mathbb{M} \subseteq \mathbb{M}_{possible}$. Every node $n \in \mathbb{M}$ has a set of edges $\mathbb{E}(n)$ that connect it to other nodes, the set of all possible edges is $\mathbb{E}_{possible}$. $\mathcal{E} : \mathbb{M} \times \mathbb{A}_{possible} \rightarrow \mathbb{E}_{possible}$ is a bijective function used to map nodes and actions to edges. Furthermore for $\mathcal{E}(n, a)$ to be valid a must be an element of $\mathbb{A}(n)$. The function $\mathcal{N}_{to} : \mathbb{E} \rightarrow \mathbb{M}$ maps an edge to the node it is pointing to while $\mathcal{N}_{from} : \mathbb{E} \rightarrow \mathbb{M}$ is used to find the node an edge is pointing from. There are, however, two different kinds of nodes, that can be distinguished by the set of their edges. The first are the expanded nodes $\mathbb{M}_{expanded} \subseteq \mathbb{M}$. For expanded nodes $n \in \mathbb{M}_{expanded}$ the set of edges $\mathbb{E}(n) = \{\mathcal{E}(n, a) \mid a \in \mathbb{A}(n)\} \neq \emptyset$. The second category are the leaf nodes $\mathbb{M}_{leaf} \subseteq \mathbb{M}$. Leaf nodes are unexpanded nodes. This means that they do not yet have any connections to other nodes. Thus $\mathbb{E}(n) = \emptyset$ for all $n \in \mathbb{M}_{leaf}$.

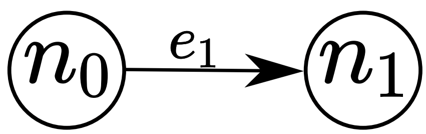


Figure 1: A vary simple MCST example

Consider the example MCST in fig 1 on page 12. The set $\mathbb{M}$ of every node in the MCTS is $\mathbb{M} = \{n_1, n_2\}$. Node n_0 has one edge e_1 pointing to n_1. Therefor $\mathbb{E}(n_0) = \{e_1\}$ and because n_1 has no edges leaving it, $\mathbb{E}(n_1) = \emptyset$. This means that $\mathbb{M}_{expanded} = \{n_0\}$ and $\mathbb{M}_{leaf} = \{n_1\}$. Furthermore $\mathcal{N}_{to}(e_1) = n_1$ and $\mathcal{N}_{from}(e_1) = n_0$. If we assume that e_1 is associated with action $a_e \in \mathbb{A}(n_0)$, than $\mathcal{E}(n_0, a_e) = e_1$

1.3.2 Quick Overview

MCTS simulations consist of four phases: selection, evaluation, expansion, and back propagation. Every simulation starts from the MCTS's root node $n_0 \in \mathbb{M}$. The selection phase of the MCTS is used to select a leaf node $n_L \in \mathbb{M}_{leaf}$. This is done by using a selection function $\sigma : \mathbb{M} \rightarrow \mathbb{M}$ defined in equation 4 on page 14. This function is used to find node n_{t+1} from node n_t for $t \in [0, L[$. n_{t+1} is selected by σ.

$$\sigma(n_t) = n_{t+1} \tag{1}$$

This means that $n_1 = \sigma(n_0)$, $n_2 = \sigma(n_1)$ .... When a leaf node is found, that leaf node is evaluated. In Alpha Zero this is achieved using the neural network described in section 1.4. The neural network evaluates the leaf node using its evaluation function $\theta : \mathbb{M} \to \mathbb{R} \times \mathbb{R}^{|\mathbb{A}_{possible}|}$. The neural network's first output is an estimation of the expected reward. The neural network's second output is the action policy $p \in \mathbb{R}^{|\mathbb{A}_{possible}|}$ that represents the advantagiousness of the actions in $\mathbb{A}_{possible}$. n_L is now expanded. Expansion works by creating nodes $\mathbb{M}_{new} = \{\mathcal{N}(\mathcal{A}(\mathcal{S}(n_L), a))|a \in \mathbb{A}(n_L)\}$ unless these nodes already exist and then redefineing $\mathbb{M} = \mathbb{M} \cup \mathbb{M}_{new}$. Furthermore, n_L's edges are redefined as $\mathbb{E}(n_L) = \{\mathcal{E}(n_L, a)|a \in \mathbb{A}(n_L)\}$. By definition $\mathcal{N}_{to}(\mathcal{E}(n_L, a) = \mathcal{N}(\mathcal{A}(\mathcal{S}(n_L), a))$ for $a \in \mathbb{A}(n_L)$. This leads to n_L being moved from $\mathbb{M}_{leaf}$ to $\mathbb{M}_{expanded}$. During back propagation the expected reward of n_L is used to update the expected reward of all nodes traversed during selection. This effectively improves the estimation of the expected reward. This four step simulation is carried out a certain amount of times. The estimation error for all estimations of the expected reward will converge to 0 as the amount of simulations increases.

1.3.3 Evaluation Basis

The MCTS's goal is to find good estimations of the reward for a certain action at a certain state. This reward estimation is $Q : \mathbb{E}_{possible} \to \mathbb{R}$. To define Q, the functions $W : \mathbb{E} \to \mathbb{R}$ and $N : \mathbb{E} \to \mathbb{N}_0$ are required. $N(e)$ is the amount of times an edge $e \in \mathbb{E}_{possible}$ has been traversed. This means how many times σ has chosen to follow the edge e to a new node. $W(e)$ is the sum of the reward computations from all $N(e)$ times the edge has been evaluated. Therefore the expected reward Q is defined as:

$$Q(e) = \frac{W(e)}{N(e)} \tag{2}$$

The fourth and last of these functions is $P : \mathbb{E} \to \mathbb{R}$. P is the policy function, it's the neural network's preliminary estimation of:

$$P(e) \approx \frac{N(e)}{\sum_{i \in \mathbb{E}(\mathcal{N}_{from}(e))} N(i)} \tag{3}$$

This function is used to guide the search to more promising edges before a lot of time is spent on simulation.

1.3.4 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 $n \in \mathbb{M}$ the function σ is used. To define σ we must first define the edge evaluation function $v : \mathbb{E} \to \mathbb{R}$. v is defined as follows:

$$v(e) = Q(e) + c_{puct}P(e) \cdot \frac{\sqrt{\Sigma_{b \in \mathbb{E}(\mathcal{N}_{from}(e))} N(b)}}{1 + N(e)} \tag{4}$$

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, for a given node $n \in \mathbb{M}$, is then defined as:

$$\sigma(n) = \mathcal{N}_{to}(\text{argmax}(\mathbb{E}(n))) \tag{5}$$

argmax returns the edge e with the largest $v(e)$. σ is run, until its output is a leaf-node $N_L \in \mathbb{M}_{leaf}$.

1.3.5 Node Evaluation and Expansion

When a leaf node $n_L \in \mathbb{M}_{leaf}$ is reached, that is not an end game node $\mathcal{S}(n_L) \notin \mathbb{G}_{done}$, the node is passed to the neural network. The neural network (see section 1.4) is used to predict the node's policy p and its value v. $\theta(n_L) = (v, p)$. p is used to create new nodes $\mathbb{M}_{new} = \{\mathcal{N}(\mathcal{A}(\mathcal{S}(n_L), a)) \ : \ a \in \mathbb{A}(n_L)\}$. The initial function outputs of the three edge functions for the edges $e \in \mathbb{E}(n_L)$, connecting n_L to $\mathbb{M}_{new}$, are.

$$N(e) = 0$$

$$W(e) = 0$$

$$P(e) = \pi(e)$$

$\pi(e)$ is the of the policy of the edge. It's defined as:

$$\pi(\mathcal{E}(n_L, a)) = p_{a+1}$$

where a is the edges action $a \in \mathbb{A}(n_L)$. $\mathbb{M}$ and $\mathbb{E}(n_L$ are then redefined as $\mathbb{M} = \mathbb{M} \cup \mathbb{M}_{new}$ and $\mathbb{E}(n_L) = \{\mathcal{E}(n_L, a) \mid a \in \mathbb{A}(n_L)\}$. $M_{expanded}$ and M_{leaf} also change according to there definition.

1.3.6 Backfill

The value v is used to update the reward prediction for all nodes $n_{[0,L]}$ traversed during the edge selection. Assuming that $\rho : \mathbb{E}_{possible} \rightarrow \{1, -1\}$ is the player taking an action at an edge $e \in \mathbb{E}_{possible}$, W and N are updated as follows for all nodes n_t with $t \in [0, L]$:

$$N(n_t) + 1 \Rightarrow N(n_t)$$
$$W(n_t) + v \cdot \rho(n_t) \cdot \rho(n_L) \Rightarrow W(n_t)$$

See fig 10 on page 305 for example Simulation.

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 [**?**]. 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 $\delta_{wb} : \mathbb{R}^n \to \mathbb{R}^m$

$$\delta_{wb}(x) = w \cdot x + b \tag{6}$$

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 $\xi_k : \mathbb{R}^{m \times n \times l} \to \mathbb{R}$ is defined as:

$$\xi_k(i) = \langle w_k, i \rangle_I + b \tag{7}$$

where $<>_I$ is the Tensor inner product defined in equation 42 on page 37. 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 $a \in [1, i-m+1]$, $b \in [1, j-n+1]$ and $c \in [1, |\mathbb{K}|]$ is defined as:

$$\Lambda(I)_{abc} = \xi_{k_c}(I[[a, a+m[, [b, b+n[, [1, |\mathbb{K}|]]) \tag{8}$$

The submatrix indexing operation $I[...]$ is defined in section 4.4 on page 37. 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 $w_k \in \mathbb{R}^{3\times3\times1}$ 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 $\Lambda(I) \in R^{2\times2\times1}$. To calculate $\Lambda(I)_{111}$, we compute the kernel operation $\xi_k(I[[1,3],[1,3],\{1\}])$

$$\Lambda_{111}\left(\begin{bmatrix} [3] & [0] & [1] & [5] \\ [2] & [6] & [2] & [4] \\ [2] & [4] & [1] & [0] \\ [3] & [0] & [1] & [5] \end{bmatrix}\right) = \begin{bmatrix} [3] & [0] & [1] \\ [2] & [6] & [2] \\ [2] & [4] & [1] \end{bmatrix} \circ \begin{bmatrix} [-1] & [0] & [1] \\ [-2] & [0] & [2] \\ [-1] & [0] & [1] \end{bmatrix} + 7 \qquad (9)$$

$$= -1\cdot3 + 0\cdot0 + 1\cdot1 - 2\cdot2 + 0\cdot6 + 2\cdot2 - 1\cdot2 + 0\cdot4 + 1\cdot1 + 7$$
$$= 4$$

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 activation 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:

$$f(x) = f_1(f_2(x)) = a(cx + d) + b = acx + ad + b = ex + g \qquad (10)$$

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, $f(x)$ becomes $f(x) = f_1(f_a(f_2(x)))$. In this neural network, three different activation functions are used: *tanh*, *softmax*, and *LeakyReLU*. These functions are defined as follows:

tanh:

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

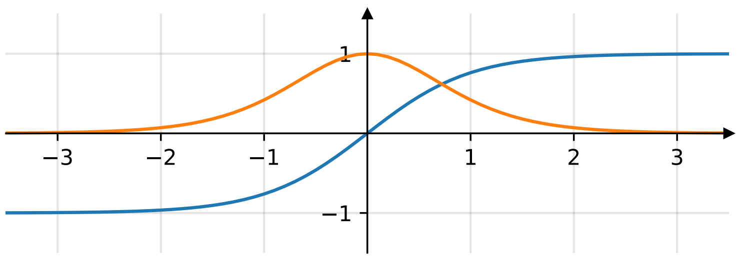


Figure 2: tanh function in blue and the tanh's derivative is in orange

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{11}$$

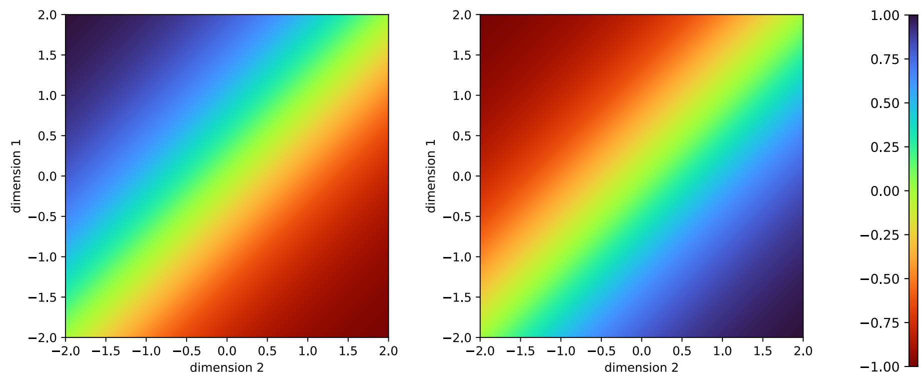
$$\frac{d}{dx}tanh(x) = sech(x)^2 \tag{12}$$

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 $i \in [1, n]$ is:

$$o_i = softmax(v)_i = \frac{e^{v_i}}{\sum_{j \in v} e^j} \tag{13}$$



(a) $i = 1$ (b) $i = 2$ scale

Figure 3: Graph of the softmax function from $\mathbb{R}^2 \rightarrow \mathbb{R}^2$. i is the index of the output dimension. Therefore, $i = 1$ referes to the output's first dimension and $i = 2$ referes to it's 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, $\frac{d}{dv_j} softmax(v)_i$, there are two possible cases.

case $j = i$:

$$\begin{aligned}
\frac{d}{dv_j}\left[\frac{e^{v_i}}{\sum_{b \in v} e^b}\right] &= \frac{\sum_{b \in v} e^b \cdot e^{v_i} - e^{v_i} \cdot e^{v_j}}{(\sum_{b \in v} e^b)^2} \\
&= \frac{e^{v_i}}{\sum_{b \in v} e^b} \cdot \frac{\left(\sum_{b \in v} e^b - e^{v_j}\right)}{\sum_{b \in v} e^b} \\
&= softmax(v)_i \cdot (1 - softmax(v)_j)
\end{aligned}$$

case $j \neq i$:

$$\begin{aligned}
\frac{d}{dv_j}\left[\frac{e^{v_i}}{\sum_{b \in v} e^b}\right] &= -\frac{e^{v_j} \cdot e^{v_i}}{\left(\sum_{b \in v} e^b\right)^2} \\
&= -softmax(v)_i \cdot softmax(v)_j
\end{aligned}$$

Therefore, the derivative of the softmax function is:

$$softmax'(v)_{ij} = \begin{cases} softmax(v)_i \cdot (1 - softmax(v)_j) & i = j \\ -softmax(v)_i \cdot softmax(v)_j & i \neq j \end{cases} \tag{14}$$

LeakyReLU:

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

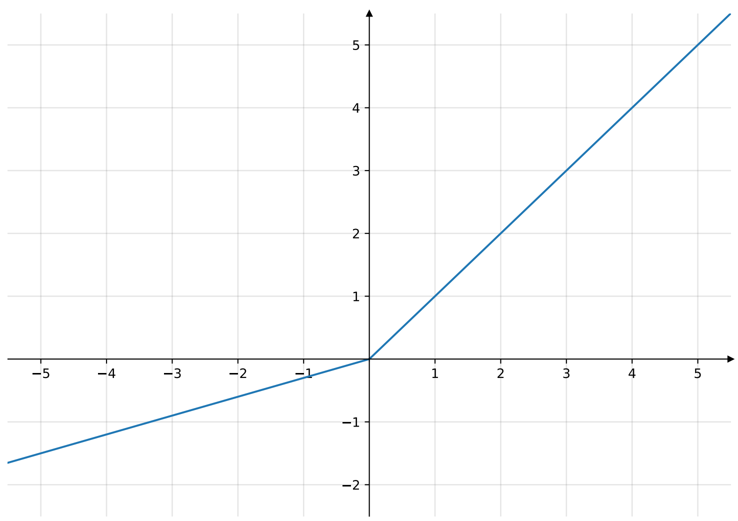


Figure 4: LeakyReLU with $c = 0.3$

$$\text{LeakyReLU}(x) = \begin{cases} x & x \geq 0 \\ x \cdot c & x < 0 \end{cases} \tag{15}$$

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

$$\text{LeakyReLU}'(x) = \begin{cases} 1 & x > 0 \\ c & x < 0 \end{cases} \tag{16}$$

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

$$g(x) = \begin{cases} 1 & x \geq 0 \\ c & x < 0 \end{cases} \tag{17}$$

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:

$$\ell = \frac{|Y_{pred} - Y_{true}|^2}{n} \tag{18}$$

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.

$$\Delta A = f'_a(A) \circ \Delta Y_j \tag{19}$$

The hadamard product $\circ$ is defined in section 4.2 on page 36.
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:

$$\Delta w = \Delta A \cdot X^T \tag{20}$$

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

$$\Delta b \sim \Delta A \tag{21}$$

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

$$\Delta Y_{j-1} = \Delta A \cdot w^T \tag{22}$$

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 22).

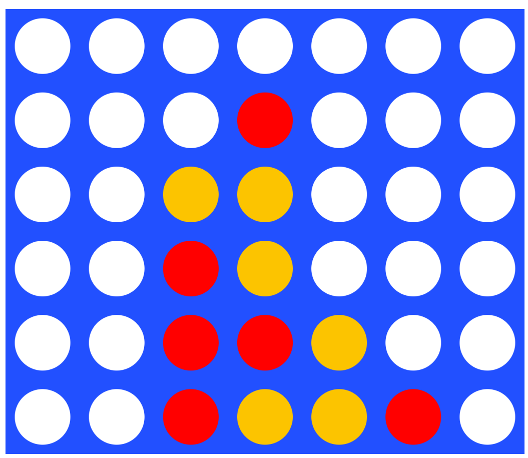


Figure 5

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.
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. 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. 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. 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 throught the softmax function to form p:

$$p = \mathrm{softmax}(p_{masked}) \tag{23}$$

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:

$$p_{a_l} = \frac{N(n, a_l)}{\sum_{a \in \mathbb{A}(n)} N(n, a)} \tag{24}$$

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

$$p_{a_n} = p_{pre_{a_n}} \tag{25}$$

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

$$P(X = a_i, n_t) = \frac{N(\mathcal{E}(n_t, a_i))}{\sum_{j \in \mathbb{A}(n_t)} N(\mathcal{E}(n_t, j))} \tag{26}$$

($\mathbb{A}(s)$ 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:

$$v_a = \begin{cases} P(X = a, \mathcal{N}(g)) & a \in \mathbb{A}(g) \\ p_{pre_a} \end{cases} \tag{27}$$

$\mathbb{A}(g)$ 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 26 on page 25.

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. [?]

2.1 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 [?]:

$$E = \frac{1}{1 + e^{(r_b - r_a)/400}} \tag{28}$$

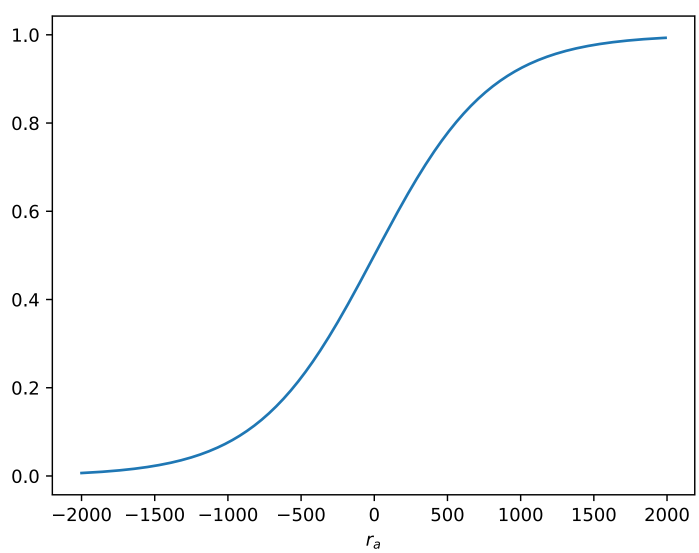


Figure 6: elo-rating win probability for $r_b = 0$

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[?]:

$$r_{n+1} = r_n + K(W - E) \tag{29}$$

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 30 where a is the player to be updated.

E the expected chance of winning, see equation 28.

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 [**?**]:

$$s_a = \frac{1}{m} \sum \begin{cases} 1 & a \text{ wins} \\ \frac{1}{2} & \text{tie} \\ 0 & a \text{ looses} \end{cases} \tag{30}$$

Assuming that this is the probability of a winning against b, a's elo-rating can be computed by solving equation 28 to r_a (fig 7 on page 29):

$$r_a = r_b - ln\left(\frac{1 - s_a}{s_a}\right) \cdot 400 \tag{31}$$

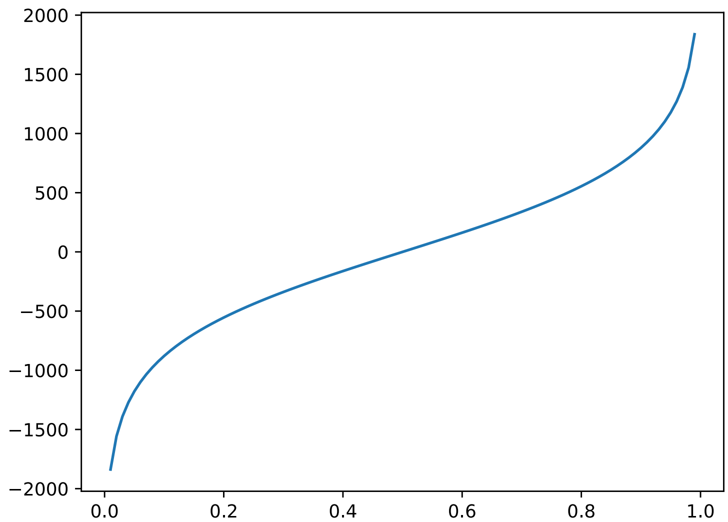


Figure 7: elo inverse function

Since a ranking of all the agents already exists (see section 1.6 on page 26), an agent's elo-rating can be computed by playing against an older version and then using equation 31 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:

$$\Delta r_{min} = -ln\left(\frac{1 - s_i}{s_i}\right) \cdot 400 \tag{32}$$

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 aginst each other, then by definition:

$$s_a + s_b = 1 \tag{33}$$

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

$$s_a \geqslant s_b \cdot \theta \tag{34}$$

(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:

$$\Delta r \geqslant -ln\left(\frac{1}{\theta}\right) \cdot 400 = \Delta r_{min} \cong 105 \tag{35}$$

Collected data shows this to be true (fig 8 on page 31). The same data shows that the average Δr is in fact roughly 408, which would equate to a θ of

$$\theta = \frac{1}{e^{\frac{-\Delta E}{400}}} \cong 2.8 \tag{36}$$

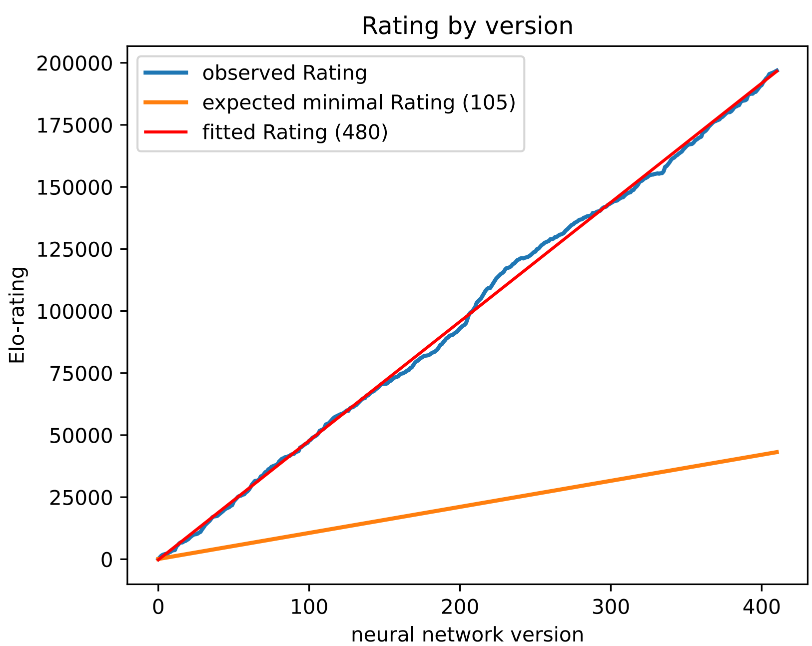


Figure 8: Elo-rating of agents based on their version along with the expected minimal rating Δr_{min} and the best fitted rating Δr.

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 cummunications 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 36), 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:

$$r = \min(\{x \in \mathbb{E} | x \geq e\}) \tag{37}$$

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 = example.json, the server would decode the json file ”example.json” shown in listing 1 on page 33. Assuming that k = [”address”, ”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.

```json
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

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 values[2]. 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 34) at which the starting player is at turn.

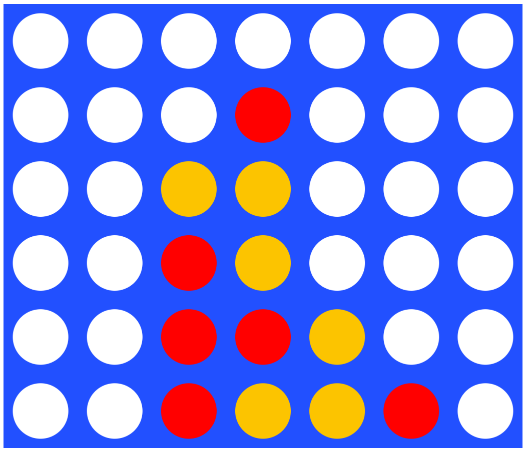


Figure 9: example game state

[2]Stored as integers 0 and 1 in memory

This state's boolean array a is:

$$a = \text{vec}\left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}\right) \frown vec\left(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}\right) \frown \langle 1 \rangle \tag{38}$$

Where *vec* is the matrix vectorization defined in section 4.5 on page 38 and $\frown$ is the vector concatination defined in section 4.6 on page 39. 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 25.

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 developped 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 tean 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 dose not create or store accounts. The player will always play against the best AI version. Thirdly, it does not have the possiblity 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 $\mathbb{B}$.

$$\mathbb{A} - \mathbb{B} = \mathbb{A} \setminus \mathbb{B} = \{x : x \in \mathbb{A} \text{ and } x \notin \mathbb{B}\} \tag{39}$$

4.2 Hadamard product

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

$$(A \circ B)_{ij} = A_{ij} \cdot B_{ij} \tag{40}$$

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

$$\langle A, B \rangle_I = \sum_{i=1}^{m} \sum_{j=1}^{n} A_{ij} B_{ij} = \sum_{i=1}^{m} \sum_{j=1}^{n} (A \circ B)_{ij} \tag{41}$$

For example consider the following 2×3 matrices:

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

4.3.2 n-dimensional Tensors

Let $A \in \mathbb{R}^{m \times \cdots}$ and $B \in \mathbb{R}^{m \times \cdots}$ be two n dimensional $m \times \ldots$ tensors with $n > 2$
Let the Inner product $\langle A, B \rangle_I : \mathbb{R}^{m \times \cdots}, \mathbb{R}^{m \times \cdots} \to \mathbb{R}$ be defined as:

$$\langle A, B \rangle_I = \sum_{i=0}^{m} \langle A_i, B_i \rangle_I \tag{42}$$

4.4 Submatrix

Let $m = m_{ijk}$ be an $m \times n \times o$ dimensional tensor.
Let $<>_{S\,x \times y,\,ab}$ be the matrix slicing opperator. $x \times y$ 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 $< m >_{S\,x \times y,\,ab}$ is defined as:

$$< m >_{S\,x \times y,\,ab} = \begin{bmatrix} [m_{ab1}, \ldots, m_{abo}] & \ldots & [m_{(a+x)b1}, \ldots, m_{(a+x)bo}] \\ \vdots & \ddots & \vdots \\ [m_{a(b+y)1}, \ldots, m_{abo}] & \ldots & [m_{(a+x)b1}, \ldots, m_{(a+x)(b+y)o}] \end{bmatrix}$$

4.5 Vectorization

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

$$\mathrm{vec}(A) = [A_{11} \ldots A_{1m}\, A_{21} \ldots A2m \ldots A_{n1} \ldots Anm]^T \tag{43}$$

Taken verbatim from:[**?**]

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

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

4.5.1 Tensors

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

$$vec(T) = T_0 \frown T_1 \frown \cdots \frown T_n \tag{44}$$

Where $\frown$ is defined defined in section 4.6 on page 39. For those who are familiar with python's numpy library, vec is equivilent to numpy.flatten. For example consider the following 3 d-Tesnor vectorization:

$$vec\left(\left[\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix}\right]\right) = \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{pmatrix}$$

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.

$$v \frown u = \begin{pmatrix} v_1 \\ \vdots \\ v_{n_v} \\ u_1 \\ \vdots \\ u_{n_u} \end{pmatrix} \tag{45}$$

Acknowledgements

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

5 Code

5.1 AlphaZeroPytorch

5.1.1 AlphaZeroPytorch.h

```
// AlphaZeroPytorch.h : Include file for standard system include
    files,
// or project specific include files.

#pragma once

#include <iostream>

// TODO: Reference additional headers your program requires here
    .
```

5.1.2 AlphaZeroPytorch.cpp

```cpp
// Entry point of training loop
//

#include "AlphaZeroPytorch.h"
#include <ai/playGame.hpp>
#include <io.hpp>
#include <chrono>
#include <thread>
#include "makeFiles.hpp"


int main(int argc, char ** argv)
{
  /*std::ofstream out("out.txt");
  std::streambuf* coutbuf = std::cout.rdbuf(); //save old buf
  std::cout.rdbuf(out.rdbuf()); //redirect std::cout to out.txt! */

  if (torch::cuda::cudnn_is_available())
  {
    std::cout << "\33[1;32mcuDNN is available\33[0m" << std::endl;
  }
  else if (torch::hasXLA())
  {
    std::cout << "\33[1;32mXLa is available\33[0m" << std::endl;
  }
  else
  {
    std::cout << "\33[1;31mWarning: cuDNN is unavailable, consider using a CUDA enabled GPU\33[0m" << std::endl;
  }
  std::vector<char*> devices = { DEVICES };
  for (auto const& device : devices)
  {
    std::cout << device << ", ";
  }

  std::cout << std::endl << "started training" << std::endl;
  createFolders();
  AlphaZero::ai::train(-1);
#if ProfileLogger
  debug::Profiler::profiler.log();
```

```
#endif
  return 0;
}
```

5.1.3 cmake.sh

```
cmake . -DCMAKE_PREFIX_PATH=/cpp/libtorch
make
```

5.1.4 CMakeLists.txt

```
# CMakeList.txt : CMake project for AlphaZeroPytorch, include source and define
# project specific logic here.
#
project(AlphaZero)

set (CMAKE_CXX_STANDARD 17)
cmake_minimum_required (VERSION 3.8)
set(CMAKE_BUILD_TYPE Debug)

message("searching for Pytorch ...")
find_package(Torch REQUIRED)

message("adding source files ...")
file(GLOB SOURCES_Files
  "include/*.cpp"
  "include/ai/*.cpp"
  "include/game/*.cpp"
)
file(GLOB HEADER_Files
  "include/*.hpp"
  "include/ai/*.hpp"
  "include/game/*.hpp"
)
file(GLOB SERVER_Files
  "include/Server/*.cpp"
)

file(GLOB OUTER_Files
  "include/*.cpp"
)

file(GLOB GAME_Files
  "include/game/*.cpp"
)

file(GLOB TEST_SOURCE
  "include/test/*.cpp"
  "include/ai/*.cpp"
  "include/game/*.cpp"
  "include/*.cpp"
)

```

```
# Add source to this project's executable.
add_executable (train "AlphaZeroPytorch.cpp" "AlphaZeroPytorch.h
    " ${SOURCES_Files} ${HEADER_Files})
add_executable (GameReplayer "Replay.cpp" "include/game/game.cpp
    " "include/config.cpp")
add_executable (runTest "test.cpp" ${TEST_SOURCE} ${
    SOURCES_Files})
add_executable (runServer "runServer.cpp" ${SOURCES_Files} ${
    SERVER_Files})
add_executable (convert "convertToJceFormat.cpp" "include/ai/
    modelWorker.cpp" "${OUTER_Files}")
add_executable (eloRaiting "doEloRaiting.cpp" ${SOURCES_Files} $
    {SERVER_Files})

target_compile_definitions(train PRIVATE cpuct_=2.0f)
target_compile_definitions(GameReplayer PRIVATE cpuct_=2.0f)
target_compile_definitions(runTest PRIVATE cpuct_=2.0f)
target_compile_definitions(runServer PRIVATE cpuct_=1.0f)
target_compile_definitions(convert PRIVATE cpuct_=2.0f)
target_compile_definitions(eloRaiting PRIVATE cpuct_=2.0f)


message("linking libs ...")
target_link_libraries(train "${TORCH_LIBRARIES}")
target_link_libraries(GameReplayer "${TORCH_LIBRARIES}")
target_link_libraries(runTest "${TORCH_LIBRARIES}")
target_link_libraries(runServer "${TORCH_LIBRARIES}")
target_link_libraries(convert "${TORCH_LIBRARIES}")
target_link_libraries(eloRaiting "${TORCH_LIBRARIES}")

message("adding includes ...")

if (WIN32)
  add_executable(showLoss "showLoss.cpp" "include/log.cpp")
  target_include_directories(showLoss PUBLIC "${CMAKE_SOURCE_DIR
    }/include")
  target_include_directories(showLoss PUBLIC "C:/Users/Julia/
    AppData/Local/Programs/Python/Python39/include")
  target_link_libraries(showLoss "C:/Users/Julia/AppData/Local/
    Programs/Python/Python39/libs/python39_d.lib")
  target_link_libraries(showLoss "${TORCH_LIBRARIES}")


  target_include_directories(train PUBLIC "D:/MyCode/CppLibs/
```

```
include")
  target_include_directories(convert PUBLIC "D:/MyCode/CppLibs/include")
  target_include_directories(GameReplayer PUBLIC "D:/MyCode/CppLibs/include")
  target_include_directories(runTest PUBLIC "D:/MyCode/CppLibs/include")
  target_include_directories(showLoss PUBLIC "D:/MyCode/CppLibs/include")
  target_include_directories(runServer PUBLIC "D:/MyCode/CppLibs/include")
  target_include_directories(eloRaiting PUBLIC "D:/MyCode/CppLibs/include")

  target_link_libraries (runServer "lib/sockpp-static_d")
  target_link_libraries (eloRaiting "lib/sockpp-static_d")
endif (WIN32)
if (UNIX)
  target_include_directories(train PUBLIC "/cpp/spdlog/include")

  target_include_directories(runServer PUBLIC "/cpp/spdlog/include")
  target_include_directories(runServer PUBLIC "/cpp/sockpp/include")

  target_include_directories(eloRaiting PUBLIC "/cpp/spdlog/include")
  target_include_directories(eloRaiting PUBLIC "/cpp/sockpp/include")

  target_include_directories(convert PUBLIC "/cpp/spdlog/include")


  target_include_directories(GameReplayer PUBLIC "/cpp/spdlog/include")

  target_include_directories(runTest PUBLIC "/cpp/spdlog/include")

  find_library(sockpp_location sockpp)
  message("sockpp is at: ${sockpp_location}")
  target_link_libraries(train "${sockpp_location}")
  target_link_libraries(runServer "${sockpp_location}")
  target_link_libraries(eloRaiting "${sockpp_location}")
```

```
endif(UNIX)


target_include_directories(train PUBLIC "${CMAKE_SOURCE_DIR}/include")
target_include_directories(runServer PUBLIC "${CMAKE_SOURCE_DIR}/include")
target_include_directories(GameReplayer PUBLIC "${CMAKE_SOURCE_DIR}/include")
target_include_directories(runTest PUBLIC "${CMAKE_SOURCE_DIR}/include")
target_include_directories(convert PUBLIC "${CMAKE_SOURCE_DIR}/include")
target_include_directories(eloRaiting PUBLIC "${CMAKE_SOURCE_DIR}/include")


message("done")
```

5.1.5 convertToJceFormat.cpp

```cpp
// Enty point of the conversion from torch archive to jce and back conversion

#include <iostream>
#include <ai/model.hpp>
#include "makeFiles.hpp"

#define FILENAME "model.jce.bin"

int main(int argc, char** argv)
{
  createFolders();
  int version=-2;
  std::cout << "What version do you want to convert -1 for current, -2 for inverse conversion: ";
  std::cin >> version;


  AlphaZero::ai::Model model("cpu");

  if (version >= 0)
  {
    model.load_version(version);
    model.jce_save_current(FILENAME);
  }
  else if (version == -1)
  {
    model.load_current();
    model.jce_save_current(FILENAME);
  }
  else if (version = -2)
  {
    model.jce_load_from_file(FILENAME);
    model.save_as_current();
  }
  else
  {
    return -1;
  }

  return 1;
}
```

5.1.6 doEloRaiting.cpp

```cpp
// rate all existing agents

#include <Server/eloClient.hpp>
#include <ai/playGame.hpp>
#include <math.h>

void evaluateAgent(int agent, int games, AlphaZero::elo::eloClient const& elo, std::shared_ptr<AlphaZero::ai::Agent> lastAgent, std::shared_ptr<AlphaZero::ai::Agent> currentAgent)
{
  std::cout << "evaluating elo for: " << agent << std::endl;

  int othersElo = elo.getElo(agent - 1);
  std::cout << "others elo is: " << othersElo << std::endl;
  AlphaZero::Game::Game* game = new AlphaZero::Game::Game();
  AlphaZero::ai::Memory* memory = new AlphaZero::ai::Memory();

  if (agent -1 > 0)
  {
    lastAgent->model->load_version(agent - 1);
  }
  if (agent > 0)
  {
    currentAgent->model->load_version(agent);
  }
  auto data = AlphaZero::ai::playGames_inThreads(game, lastAgent.get(), currentAgent.get(), memory, 2, 1, games, "eloEvaluation");

  int wins = data[currentAgent.get()];
  int losses = data[lastAgent.get()];
  int ties = games - wins - losses;

  float score = ((float)wins + 0.5f * (float)ties)/((float)games);
  if (score > 0.99)
  {
    score = 0.99f;
  }
  float Relo = (float)othersElo - log((1 - score) / score) * 400;
  elo.setElo(agent, (int)Relo);
```

```

  std::cout << wins << " wins, " << ties << " ties and " <<
    losses << " losses" << std::endl;
  std::cout << "win Ratio is : " << score << std::endl;
  std::cout << "new rating is: " << Relo << std::endl << std::
    endl;

  delete game;
  delete memory;
}

int main()
{
  std::vector<char*> devices = { DEVICES };
  std::shared_ptr<AlphaZero::ai::Agent> lastAgent = std::
    make_shared<AlphaZero::ai::Agent>(devices);
  std::shared_ptr<AlphaZero::ai::Agent> currentAgent = std::
    make_shared<AlphaZero::ai::Agent>(devices);

  AlphaZero::elo::eloClient elo;
  std::cout << elo.setElo(0, 100) << std::endl;

  int agent = 409;
  while (true)
  {
    evaluateAgent(agent, 40, elo, lastAgent, currentAgent);
    agent++;
  }
  return 1;
}
```

5.1.7 makeFiles.hpp

```cpp
#pragma once

#ifndef UNIX
#include <filesystem>
#endif
#include <jce/string.hpp>
#include <string>

void createFolder(std::string str)
{
  createFolder(str.c_str());
}

void createFolder(char name[])
{
#ifndef UNIX
  std::filesystem::create_directories(name);
#else
  const char* foo = "mkdir -p ";
  char* full_text = new char[100];
  strcpy(full_text, foo);
  strcat(full_text, name);
  system(full_text);
#endif
}

void inline createFolders()
{
  char folder[100];

  sprintf(folder, "models/run_%d", runVersion);
  createFolder(folder);

  sprintf(folder, "memory/run_%d", runVersion);
  createFolder(folder);

  sprintf(folder, "logs/c++");
  createFolder(folder);

  sprintf(folder, "logs/games");
  createFolder(folder);
}
```

5.1.8 Replay.cpp

```cpp
// Replay a game (unused)

#include <iostream>
#include <io.hpp>
#include <config.hpp>

int main(int argc, char ** argv)
{
#if SaverType == 2
  auto saver = AlphaZero::io::ActionsOnly::GameSaver();
#elif SaverType == 1
  auto saver = AlphaZero::io::FullState::GameSaver();
#endif
#if SaverType != 0
  saver.load("test.bin");
  saver.ConsoleReplay(0);
#endif
  return 1;
}
```

5.1.9 runServer.cpp

```cpp
// Enty point of the Ai server

#include <config.hpp>
#include <Server/server.hpp>
#include <game/game.hpp>

int main()
{
#if OPSMode == 1
  AlphaZero::Server::TCPServer server;
  server.mainLoop();

#elif OPSMode == 2
  std::cout << "\33[1;31mUsing Test Server! \n\tset OPSMode to 1
    for server if not testing\33[0m" << std::endl;
  AlphaZero::Server::TestServer server(PORT);
  server.mainLoop();
#endif
}
```

5.1.10 showLoss.cpp

```cpp
// ehhhh. (not the fogiest clue)

#include <iostream>
#include <log.hpp>
#include <python.h>

int main(int argc, char** argv)
{
  debug::log::lossLogger log("logs/games/loss.bin");

  return 1;
}
```

5.1.11 test.cpp

```cpp
// run the test script

#include <test/testSuit.hpp>
#include <jce/save.hpp>
#include <jce/load.hpp>
#include "makeFiles.hpp"
#include <vector>


int main(int argc, char** argv)
{
  createFolders();
  AlphaZero::test::runTests();

  /*std::vector<int> count;
  std::ifstream in("models/run_1/iterationCounter.jce");
  jce::load(in, count);
  in.close();
  for (auto const& val : count)
  {
    std::cout << val << ", ";
  }*/
  return 1;
}
```

5.1.12 include

5.1.12.1 config.hpp

```
#pragma once
#include <log.hpp>
#include <bitset>
#include <mutex>

#ifdef unix
#define UNIX
#endif

//#ifdef UNIX
//#define DEVICES "cuda:0"
//#endif
//#ifndef UNIX
//#define DEVICES "cpu"
//#endif
#define DEVICES "cpu"

#define OPSMode 1

extern std::mutex console_mutex;
extern std::mutex rand_mutex;

/*
+----------+---------------+
| OPSMode  | Description   |
+----------+---------------+
| 1        | Run Server    |
+----------+---------------+
| 2        | Run Tester    |
+----------+---------------+
*/

#define GameChecksLegalMoved true // the game will check if a move is legal not neded for training
#define stateSize 84
#define Training true
#define DEBUG false

#define U_computation(edge) (this->cpuct * edge.P * std::sqrt((float)Nb) / (float)(1 + edge.N))

```

```
// runn setting
#define runVersion 1
#define loadVersion -1

// Net settings
#define MaxQuDefault -99999
#define reg_const 0.0001
#define learningRage 0.1
#define Momentum 0.9

// simulation setting
#define MCTSSimulations 50
//#define cpuct_ 2.0f
#define ProbabiliticMoves 10
#define Alpha 0.9
#define EPSILON 0.2f

// memory setting
#define memory_size 30000

// self play
#define EPOCHS 1
#define GEN_THREADS 60
#define probabilitic_moves 10 // how manny moves are prabilistic
    in the begining of the game to aid in exploration

// training
#define Training_loops 20
#define Training_batch 256
#define Training_epochs 5

// turney
#define Turnement_probabiliticMoves 2
#define TurneyEpochs 1
#define TurneyThreads 20
#define scoringThreshold 1.3

// console
#define RenderTrainingProgress false
#define RenderGenAndTurneyProgress false

// Saving
#define SaverType 0
/* +------------+----------------------------+
```

```
  | SaverType | Description                  |
  +-----------+------------------------------+
  | 0         | no Saver                     |
  +-----------+------------------------------+
  | 1         | save full state to file      |
  +-----------+------------------------------+
  | 2         | Save taken Actions to file   |
  +-----------+------------------------------+*/
#define SaverIntSize 1


typedef std::bitset<stateSize> IDType;
```

5.1.12.2 config.cpp

```
#include "config.hpp"

std::mutex console_mutex;
std::mutex rand_mutex;
```

5.1.12.3 io.hpp

```cpp
#pragma once
#include <jce/load.hpp>
#include <jce/save.hpp>
#include "config.hpp"

// classes that record games and stores them to file.

namespace AlphaZero
{
    namespace io
    {
        namespace FullState
        {
            class GameSaver
            {
              // list of states the game passed though
            public: std::list<std::list< std::shared_ptr<Game::GameState>>> states;
              // add a state to the states list last list
            public: void addState(std::shared_ptr<Game::GameState> state);
              // add a game to the states list
            public: void addGame();
              // remove all games
            public: void clear();
              // save the saver to file
            public: void save(char filename[]);
              // load a saver from file
            public: void load(char filename[]);

            // replay game in console (debug)
            public: void ConsoleReplay(int itx);
            };
        }
        namespace ActionsOnly
        {
            class GameSaver
            {
              // list of action taken during game pass
            public: std::list<std::list<unsigned int>> states;
              // add an action to the current game
            public: void addState(int);
              // add a game to the list of games
```

```
            public: void addGame();
              // remove all games
            public: void clear();
              // save saver to file
            public: void save(char filename[]);
              // load saver from file
            public: void load(char filename[]);

            // replay game in console (debug)
            public: void ConsoleReplay(int idx);
            };
        }
    }
}

inline void AlphaZero::io::FullState::GameSaver::addState( std::
    shared_ptr<Game::GameState> state)
{
    this->states.back().push_back(state);
}

inline void AlphaZero::io::FullState::GameSaver::addGame()
{
    this->states.push_back( std::list<std::shared_ptr<Game::
    GameState>>());
}

inline void AlphaZero::io::FullState::GameSaver::clear()
{
    this->states.clear();
}

inline void AlphaZero::io::FullState::GameSaver::save(char
    filename[])
{
    std::ofstream fout;
    fout.open(filename, std::ios::binary);
    if (fout.is_open())
    {
        jce::save(fout, this->states);
        fout.close();
    }
    else
    {
        throw "Game saver file not opend.";
```

```
    }
}

inline void AlphaZero::io::FullState::GameSaver::load(char filename[])
{
    std::ifstream infile(filename, std::ios::binary);

    if (infile.is_open())
    {
        jce::load(infile, this->states);
        infile.close();
    }
    else
    {
    throw "Game saver file not opend.";
    }
}

inline void AlphaZero::io::FullState::GameSaver::ConsoleReplay(int idx)
{
    for (auto const& state : *std::next(this->states.begin(), idx))
    {
        state->render();
    }
}


inline void AlphaZero::io::ActionsOnly::GameSaver::addState(int actions)
{
    this->states.back().push_back(actions);
}

inline void AlphaZero::io::ActionsOnly::GameSaver::addGame()
{
    this->states.push_back(std::list<unsigned int>());
}

inline void AlphaZero::io::ActionsOnly::GameSaver::clear()
{
    this->states.clear();
}
```

```
inline void AlphaZero::io::ActionsOnly::GameSaver::save(char
    filename[])
{
    std::ofstream file(filename, std::ios::binary);
    if (file.is_open())
    {
        jce::save(file, this->states);
        file.close();
    }
    else
    {
        throw "Game saver file not opend.";
    }
}

inline void AlphaZero::io::ActionsOnly::GameSaver::load(char
    filename[])
{
    std::ifstream file(filename, std::ios::binary);
    if (file.is_open())
    {
        jce::load(file, this->states);
        file.close();
    }
    else
    {
        throw "Game saver file not opend.";
    }
}

inline void AlphaZero::io::ActionsOnly::GameSaver::ConsoleReplay
    (int idx)
{
    Game::Game game = Game::Game();

    for (int action : *std::next(this->states.begin(), idx))
    {
        game.render();
        game.takeAction(action);
    }
    game.render();
}
```

5.1.12.4 log.hpp

```cpp
#pragma once
#define threads 0

//logging
#define MainLogger true
#define MCTSLogger false
#define MemoryLogger false
#define ProfileLogger false
#define ModelLogger true
#define LossLogger true


#include <unordered_map>
#include <spdlog/sinks/basic_file_sink.h>
//#include <memory>
#if ProfileLogger
#include "timer.hpp"
#endif
#include <stdio.h>
#include <chrono>
#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger || LossLogger)

// logging header

namespace debug {
#if ProfileLogger
    // Profiler used to profile the AI
  namespace Profiler {
    class MCTSProfiler {
      // its a timer.
    private: utils::Timer timer;
      // times by processid.
    public: std::unordered_map<unsigned int, double> times;
      // how long the profiler spent without an assigned profiler.
    private: double rest;

      // whether the Object has not switchOperation is called for the first time
    private: bool first = true;
      // false if the Profiler has an assigned profiler. (else true)
```

```
    private: bool toRest = false;
      // the id of the current process
    private: unsigned int currentTime;

      // switch to a diferant process
    public: void switchOperation(unsigned int id);
      // stop profiling the current process and start running on
     rest.
    public: void stop();
      // print log to profile logger
    public: void log();
      // print to a diferant logger.
    public: void log(std::shared_ptr<spdlog::logger> logger);
    };
    extern debug::Profiler::MCTSProfiler profiler;
  };
#endif
  namespace log {
      // create a logger of a certain name on a certain file
    std::shared_ptr<spdlog::logger> createLogger(const char*
   name, const char* file);
    template<typename T>
      // log vector to logger. and put some text befor every
   line
    void logVector(std::shared_ptr<spdlog::logger> logger, std::
   vector<T>, char out[]);
      // log vector of intagers to logger
    void logVector(std::shared_ptr<spdlog::logger> logger, std::
   vector<int>);
      // log vector of floats to logger.
    void logVector(std::shared_ptr<spdlog::logger> logger, std::
   vector<float>);

    class lossLogger
    {
    public: lossLogger();
    public: lossLogger(const char file[]);

    protected: std::vector<std::vector<std::pair<float, float>>>
     vals;

    public: void addValue(const float val, const float poly);
    public: void addValue(const std::pair<float, float>& val);
    public: void newBatch();
    public: void save(const char file[]);
```

```
    public: std::pair<float, float> operator[](std::pair<size_t, size_t> idx) const;
    public: std::vector<std::pair<float, float>> operator[](size_t idx) const;
    public: bool operator==(const lossLogger&);
    };

#if MainLogger
    extern std::shared_ptr<spdlog::logger> mainLogger;
#endif
#if MCTSLogger
    extern std::shared_ptr<spdlog::logger> MCTS_Logger;
#endif
#if MemoryLogger
    extern std::shared_ptr<spdlog::logger> memoryLogger;
#endif
#if ProfileLogger
    extern std::shared_ptr<spdlog::logger> profileLogger;
#endif
#if ModelLogger
    extern std::shared_ptr<spdlog::logger> modelLogger;
#endif
#if LossLogger
    extern lossLogger _lossLogger;
#endif
  }
}

inline std::shared_ptr<spdlog::logger> debug::log::createLogger(const char* name, const char* file) {
  // Create a daily logger - a new file is created every day on 2:30am
  int success = remove(file);
  auto logger = spdlog::basic_logger_mt(name, file);
  return logger;
}

inline debug::log::lossLogger::lossLogger()
{
  this->newBatch();
}

inline void debug::log::lossLogger::addValue(const float a, const float b)
```

```cpp
{
  std::pair<float, float> data = { a, b };
  this->addValue(data);
}

inline void debug::log::lossLogger::addValue(const std::pair<
    float, float>& val)
{
  this->vals.back().push_back(val);
}

inline void debug::log::lossLogger::newBatch()
{
  this->vals.push_back({});
}

inline std::pair<float, float> debug::log::lossLogger::operator
    [](std::pair<size_t, size_t> idx) const
{
  return this->vals[idx.first][idx.second];
}

inline std::vector<std::pair<float, float>> debug::log::
    lossLogger::operator[](size_t idx) const
{
  return this->vals[idx];
}

template<typename T>
inline void debug::log::logVector(std::shared_ptr<spdlog::logger
    > logger, std::vector<T> vec, char out[])
{
  logger->info(out, (vec[0]), (vec[1]), (vec[2]), (vec[3]),
    (vec[4]), (vec[5]), (vec[6]));
  logger->info(out, (vec[7]), (vec[8]), (vec[9]), (vec[10]),
    (vec[11]), (vec[12]), (vec[13]));
  logger->info(out, (vec[14]), (vec[15]), (vec[16]), (vec[17]),
    (vec[18]), (vec[19]), (vec[20]));
  logger->info(out, (vec[21]), (vec[22]), (vec[23]), (vec[24]),
    (vec[25]), (vec[26]), (vec[27]));
  logger->info(out, (vec[28]), (vec[29]), (vec[30]), (vec[31]),
    (vec[32]), (vec[33]), (vec[34]));
  logger->info(out, (vec[35]), (vec[36]), (vec[37]), (vec[38]),
    (vec[39]), (vec[40]), (vec[41]));
}
```

```
inline void debug::log::logVector(std::shared_ptr<spdlog::logger> logger, std::vector<int> vec)
{
  char out[] = "Action vals are: {:3d}, {:3d}, {:3d}, {:3d}, {:3d}, {:3d}, {:3d}";
  logVector(logger, vec, out);
}

inline void debug::log::logVector(std::shared_ptr<spdlog::logger> logger, std::vector<float> vec)
{
  char out[] = "Action vals are: {:1.2f}, {:1.2f}, {:1.2f}, {:1.2f}, {:1.2f}, {:1.2f}, {:1.2f}";
  logVector(logger, vec, out);
}
#if ProfileLogger
inline void debug::Profiler::MCTSProfiler::switchOperation(unsigned int id)
{
  this->stop();
  this->first = false;
  this->toRest = false;
  this->timer.reset();
  this->currentTime = id;
}

inline void debug::Profiler::MCTSProfiler::stop()
{

  if (!this->first) {
    if (toRest) {
      this->rest = this->rest + this->timer.elapsed();
    }
    double currentNum;
    if (this->times.count(currentTime) == 0) {
      currentNum = 0.0f;
    }
    else {
      currentNum = this->times.at(this->currentTime);
    }
    double res = currentNum + this->timer.elapsed();
    this->times.insert_or_assign( currentTime, res);
  }
  this->toRest = true;
```

```
}

inline void debug::Profiler::MCTSProfiler::log() {
  this->log(debug::log::profileLogger);
}

inline void debug::Profiler::MCTSProfiler::log(std::shared_ptr<spdlog::logger> logger)
{
  logger->info("using {} threads", threads);
  logger->info("run Info:");
  logger->info("");

  // this is only true on the computers im using as im just entering the infomation here to save time
#ifdef WIN32
  logger->info("os: Windows 10");
  logger->info("CPU: Intel(R) Core(TM) i5-8350U CPU @ 1.70 GHz ");
  logger->info("GPU: None");
  logger->info("memory: 8 GB");
#else
  logger->info("OS: Ubuntu 18.04");
  logger->info("CPU: ??");
  logger->info("GPU: Nvidia P4 cuda 11.4");
  logger->info("memory: 7.8 Gb");
#endif

  logger->info("==================================================================================================");
  for (auto const& pair : this->times) {
    logger->info("Profiler time id {} took {} s", pair.first, pair.second);
  }
  logger->info("everything else took: {}", this->rest);

  logger->info("");
  logger->info("0 : MCTS and NN forward");
  logger->info("3 : Game Stuff");
  logger->info("4 : Memory shuffeling");
  logger->info("5 : NN Backward");
}
#endif
#endif
```

5.1.12.5 log.cpp

```cpp
#include "log.hpp"
#include <stdio.h>
#include <jce/save.hpp>
#include <jce/load.hpp>



#if MainLogger
std::shared_ptr<spdlog::logger> debug::log::mainLogger = debug::log::createLogger("mainLogger", "logs/c++/mainLogger.log");
#endif
#if MCTSLogger
std::shared_ptr<spdlog::logger> debug::log::MCTS_Logger = debug::log::createLogger("MCTS_Logger", "logs/c++/MCTS_Logger.log");
#endif
#if MemoryLogger
std::shared_ptr<spdlog::logger> debug::log::memoryLogger = debug::log::createLogger("memoryLogger", "logs/c++/memoryLogger.log");
#endif
#if ProfileLogger
std::shared_ptr<spdlog::logger> debug::log::profileLogger = debug::log::createLogger("profileLogger", "logs/c++/profileLogger.log");

debug::Profiler::MCTSProfiler debug::Profiler::profiler = debug::Profiler::MCTSProfiler();
#endif
#if ModelLogger
std::shared_ptr<spdlog::logger> debug::log::modelLogger = debug::log::createLogger("ModelLogger", "logs/c++/ModelLogger.log");
#endif
#if LossLogger
debug::log::lossLogger debug::log::_lossLogger = debug::log::lossLogger();
#endif

debug::log::lossLogger::lossLogger(const char file[])
{
  std::ifstream in(file, std::ios::binary);
  if (in.is_open())
```

```
  {
    jce::load(in, this->vals);
  }
  else
  {
    std::cout << "\33[31;1mFailed to load lossLogger from " << file << "\33[0m" << std::endl;
  }
  in.close();
}

void debug::log::lossLogger::save(const char file[])
{
  std::ofstream out (file, std::ios::binary);
  if (out.is_open())
  {
    jce::save(out, this->vals);
  }
  else
  {
    std::cout << "\33[31;1mFailed to save lossLogger to " << file << "\33[0m" << std::endl;
  }
  out.close();
}

bool debug::log::lossLogger::operator==(const lossLogger& other)
{
  if (other.vals.size() == this->vals.size())
  {
    for (size_t batch = 0; batch < other.vals.size(); batch++)
    {
      if (other.vals[batch].size() == this->vals[batch].size())
      {
        for (size_t idx = 0; idx < other.vals[batch].size(); idx++)
        {
          if (other[idx] != (*this)[idx])
          {
            return false;
          }
        }
      }
      else
      {
```

```
        return false;
      }
    }
    return true;
  }
  return false;
}
```

5.1.12.6 timer.hpp

```
#pragma once
#include <iostream>
#include <chrono>

namespace utils {
    class Timer
    {
    public:
        Timer();
        void reset();
        double elapsed() const;

    private:
        typedef std::chrono::high_resolution_clock clock_;
        typedef std::chrono::duration<double, std::ratio<1> >
    second_;
        std::chrono::time_point<clock_> beg_;
    };
};


inline utils::Timer::Timer()
{
    this->beg_ = clock_::now();
}

inline void utils::Timer::reset()
{
    this->beg_ = clock_::now();
}

inline double utils::Timer::elapsed() const
{
        return std::chrono::duration_cast<second_>(clock_::now()
    - beg_).count();
}
```

5.1.12.7 ai

5.1.12.7.1 agent.hpp

```cpp
#pragma once
#include <ai/modelSynchronizer.hpp>
#include <ai/memory.hpp>
#include <jce/vector.hpp>
#include "utils.hpp"
#include <thread>

namespace AlphaZero {
  namespace ai {
    // Alpha zero agent used to call the MCTS and train the Neural Network.
    class Agent {
      // MCTS tree by thread id. (thread the game is running in)
    public: std::unordered_map<size_t, std::shared_ptr<MCTS>> tree;
      // pointer to the model synchronizer that handles the model.
    public: std::unique_ptr<AlphaZero::ai::ModelSynchronizer> model;

      // create an agent and initialize the models on multiple devices
    public: Agent(std::vector<char*> devices);

      // get the MCTS tree for the current thread
    public: MCTS* getTree();

      // remove all MCTS trees
    public: void reset();

      // get action and action values from the MCTS. Also call simulations and evaluate NN
    public: std::pair<int, std::pair<std::vector<int>, float>> getAction(std::shared_ptr<Game::GameState> state, bool proabilistic);

      // run a single MCTS simulation
    public: void runSimulations(Node*, MCTS* tree);

      // evaluate and expand a leaf node.
    private: float evaluateLeaf(Node*, MCTS* tree);
```

```cpp

      // train the neural network
    public: void fit(std::shared_ptr<Memory> memory, unsigned short iteration);

      // evaluate the neural network
    public: std::pair<float, std::vector<float>> predict(std::shared_ptr<Game::GameState> state);
    public: void predict(ModelData* data);
    public: void predict(std::list<ModelData*> data);

      // get action deterministicly (largest edge.N)
    private: std::pair<int, std::pair<std::vector<int>, float>> derministicAction(Node* node);

      // get action probabilistically (random waited by edge.N)
    private: std::pair<int, std::pair<std::vector<int>, float>> prabilisticAction(Node* node);
    };
    // call simulations untill MCTSIter == MCTSSimulations (config.hpp)
    void runSimulationsCaller(AlphaZero::ai::Agent* agent, Node* node, MCTS* tree);
  }
}

inline AlphaZero::ai::MCTS* AlphaZero::ai::Agent::getTree()
{
  auto a = std::hash<std::thread::id>{}(std::this_thread::get_id());
  if (this->tree.count(a))
  {
    return this->tree[a].get();
  }
  else
  {
    auto tree = std::make_shared<MCTS>();
    this->tree.insert({ a, tree });
    return tree.get();
  }
}

inline void AlphaZero::ai::Agent::reset()
{
  this->tree.clear();
```

```
}

inline void AlphaZero::ai::Agent::runSimulations(Node* node, MCTS* tree)
{
  std::pair<Node*, std::list<Edge*>> serchResults = tree->moveToLeaf(node);
  float val = this->evaluateLeaf(serchResults.first, tree);
  tree->backFill(serchResults.second, serchResults.first, val);
  tree->addMCTSIter();
}

inline void AlphaZero::ai::runSimulationsCaller(AlphaZero::ai::Agent* agent, Node* node, MCTS* tree)
{
  while (tree->MCTSIter < MCTSSimulations) {
    agent->runSimulations(node, tree);
  }
}

inline float AlphaZero::ai::Agent::evaluateLeaf(Node* node, MCTS* tree)
{
  if (!node->state->done){
    std::shared_ptr<Game::GameState> nextState;
    Node* nextNode;
    auto data = ModelData(node);
    this->predict(&data);
    for (auto& action : node->state->allowedActions) {
      nextState = node->state->takeAction(action);
      nextNode = tree->addNode(nextState);
      Edge newEdge = Edge(nextNode, node, action, data.polys[action].item<float>()); //the last is the prob
      node->addEdge( action, newEdge);
    }
    return data.value;
  }
  return (float)std::get<0>(node->state->val);
}

inline void AlphaZero::ai::Agent::fit(std::shared_ptr<Memory> memory, unsigned short run)
{
  std::cout << "\33[35;1mretraining\33[0m" << std::endl;
  for (int idx = 0; idx < Training_loops ; idx++) {
```

```cpp
#if RenderTrainingProgress
    jce::consoleUtils::render_progress_bar((float)idx / (float)Training_loops);
#endif
    auto batch = Model::getBatch(memory, Training_batch);
    for (size_t trainingEpoch = 0; trainingEpoch < Training_epochs; trainingEpoch++)
      this->model->fit(batch, run, idx);

#if LossLogger
    debug::log::_lossLogger.newBatch();
#endif
  }
  this->model->synchronizeModels();
#if LossLogger
  debug::log::_lossLogger.save("logs/games/loss.bin");
#endif
#if RenderTrainingProgress
  jce::consoleUtils::render_progress_bar(1.0f, true);
#endif
}

inline std::pair<float, std::vector<float>> AlphaZero::ai::Agent::predict(std::shared_ptr<Game::GameState> state)
{
  auto preds = this->model->predict(state);
  float& val = preds.first;
  std::vector<float> polys = std::vector<float>(action_count);

  c10::Device device ("cpu");

  torch::Tensor mask = torch::ones(
    { 1, action_count },
    c10::TensorOptions().device(c10::Device("cpu")).dtype(at::kBool)
  );

  for (auto idx : state->allowedActions)
  {
    mask[0][idx] = false;
  }

  torch::Tensor out = torch::softmax(torch::masked_fill(preds.second.cpu(), mask, -1000.0f), 1);

```

```cpp
  for (auto const& idx : state->allowedActions) {
    polys[idx] = out[0][idx].item<float>();
  }

  return { val, polys };
}

inline void AlphaZero::ai::Agent::predict(ModelData* data)
{
  this->model->addData(data);
}

inline void AlphaZero::ai::Agent::predict(std::list<ModelData*> data)
{
  this->model->predict(data);
}

inline std::pair<int, std::pair<std::vector<int>, float>> AlphaZero::ai::Agent::derministicAction(Node* node)
{
  int action = 0;
  unsigned int max_N = 0;
  unsigned int sum = 0;
  std::vector<int> probs = jce::vector::gen(action_count, 0);
  for (auto const& iter : node->edges) {
    if (iter.second.N > max_N) {
      max_N = iter.second.N;
      action = iter.first;
    }
    probs[iter.second.action] = iter.second.N;
  }
  return { action, { probs, node->edges[action].Q } };
}

inline std::pair<int, std::pair<std::vector<int>, float>> AlphaZero::ai::Agent::prabilisticAction(Node* node)
{
  int action = -1;
  int idx = 0;
  unsigned int sum = 0;
  std::vector<int> probs = jce::vector::gen(action_count, 0);

  for (auto const& iter : node->edges) {
    probs[iter.second.action] = (iter.second.N);
```

```
  }
  auto action_probs = (rand() % ai::getSumm(probs));

  for (auto const& val : probs) {
    action_probs -= val;
    if (action_probs < 0) {
      action = idx;
      break;
    }
    idx++;
  }
  return { action, { probs, node->edges[action].Q } };
}
```

5.1.12.7.2 agent.cpp

```
#include "agent.hpp"
#include <stdlib.h>

AlphaZero::ai::Agent::Agent(std::vector<char*> devices)
{
  this->tree = {};
  this->model = std::make_unique<ModelSynchronizer>(devices);
}

std::pair<int, std::pair<std::vector<int>, float>> AlphaZero::ai
    ::Agent::getAction(std::shared_ptr<Game::GameState> state,
    bool proabilistic)
{
#if ProfileLogger
  debug::Profiler::profiler.switchOperation(0);
#endif
  auto tree = this->getTree();
  tree->MCTSIter = 0;
  Node* node = tree->addNode(state);
#if threads > 0
  std::vector<std::thread> threadvec;
  for (int i = 0; i < threads; i++) {
    threadvec.push_back(std::thread(runSimulationsCaller, this,
    node));
  }
#endif
  runSimulationsCaller(this, node, tree);
#if threads > 0
  for (auto& thread : threadvec) {
    thread.join();
  }
#endif
  try {
#if ProfileLogger
    debug::Profiler::profiler.switchOperation(3);
#endif
    if (proabilistic) {
      return this->prabilisticAction(node);
    }
    else {
      return this->derministicAction(node);
    }
  }
```

```
  catch (const std::exception& ex) {
    std::cerr << "\33[31;1mError in Agent::getAction\33[0m" << std::endl;
    std::cerr << ex.what() << std::endl;
    throw ex;
  }
}
```

5.1.12.7.3 MCTS.hpp

```cpp
#pragma once
#include <mutex>
#include <game/game.hpp>
#include <jce/vector.hpp>

// remove one mutex - the Node mutex


namespace AlphaZero {
  namespace ai {
    class Node;

    /*Class Represienting the action connecting 2 nodes togather. It handles all The MCTS relevant variables and the mutex for
    * parallization
    */
    class Edge {
      // The number of times the Edge was traversed
    public: int N = 0;
        // the probability initialized by the NN
    public: float P = 0;
        // the action asociated with the action
    public: int action = 0;
        // the amount of times this lead to a win
    public: float W = 0;
        // the win probability
    public: float Q = 0;

      // the node the edge leads from
    public: Node* outNode;

      // the node the edge leads to
    public: Node* inNode;

      // build an edge betwean two nodes on a certain action with a certain prediction p
    public: Edge(Node* outNode, Node* inNode, int action, float p);

      // debugging function used find faulty initializations (depricated)
    public: Edge();
```

```cpp

    // increase N by 1
   public: void traverse();
   };

    // class representing a node in a MCTS graph
   class Node {

    // the game state the node is asociated with
   public: std::shared_ptr<Game::GameState> state;

    // the map of actions to edges that can be taken at the node
   public: std::unordered_map<int, Edge> edges;

    // create the node on a certain game state
   public: Node(std::shared_ptr<Game::GameState>);

    // check if the node is a leaf (edges.size() == 0)
   public: bool isLeaf();

    // add an edge to the map of edges.
   public: void addEdge(int id, Edge& edge);
   };

    // generate a list with a float for every action in the game. the list is 0
    // if the action is illegal and the computed Q by the mcts if not. the index in
    // the list is the action number
   std::vector<float> getQ(Node*);

    // class that handles MCTS simulations
   class MCTS {
    // counter used to count the ammount of simulations performed by the MCTS.
   public: unsigned short MCTSIter = 0;
    // map of nodes by game state.
   private: std::unordered_map<IDType, std::unique_ptr<Node>> MCTS_tree;

        // add 1 to MCTSIter
   public: void addMCTSIter();
    // create a MCTS object and initialize it (do nothing)
   public: MCTS();
```

```cpp

      // save cpuct to the model from the config file.
    public: float cpuct = cpuct_;

      // start from the given node and follow the procedure outlined in the paper to
      // move to a leaf node
    public: std::pair<Node*, std::list<Edge*>> moveToLeaf(Node*);

      // back up from the leaf node and update the values appropriatly.
    public: void backFill(std::list<Edge*>&, Node* leaf, float val);

      // get the node by Game id
    public: Node* getNode(IDType);

      // get a node by game state (will automatically create nodes if necesary)
    public: Node* addNode(std::shared_ptr<Game::GameState> state);

      // remove all nodes from the graph.
    public: void reset();
    };
  }
}

inline std::vector<float> AlphaZero::ai::getQ(Node* node)
{
  std::vector<float> data = jce::vector::gen(42, 0.0f);
  for (auto const& pos : node->edges)
  {
    data[pos.first] = pos.second.Q;
  }
  return data;
}

inline void AlphaZero::ai::MCTS::addMCTSIter()
{
  // this->MCTSIterMutex.lock();
  MCTSIter++;
  // this->MCTSIterMutex.unlock();
}
```

```cpp

inline bool AlphaZero::ai::Node::isLeaf()
{
  return this->edges.size() == 0;
}

inline void AlphaZero::ai::Node::addEdge(int id, Edge& edge)
{
  // this->lock.lock();
  this->edges.insert({ id, edge });
  // this->lock.unlock();
}

inline AlphaZero::ai::MCTS::MCTS(){}

inline AlphaZero::ai::Node* AlphaZero::ai::MCTS::addNode(std::
    shared_ptr<Game::GameState> state)
{
  if (this->MCTS_tree.count(state->id()) == 0) {

     // this->NodeInsersionMutex.lock();
     this->MCTS_tree.insert({ state->id(), std::make_unique<Node
    >(state)});
     // this->NodeInsersionMutex.unlock();
  }
  return this->getNode(state->id());
}

inline void AlphaZero::ai::MCTS::reset()
{
  // this->NodeInsersionMutex.lock();
  this->MCTS_tree.clear();
  // this->NodeInsersionMutex.unlock();
}

inline AlphaZero::ai::Node* AlphaZero::ai::MCTS::getNode(IDType
    key)
{
  return this->MCTS_tree[key].get();
}


inline void AlphaZero::ai::Edge::traverse()
{
  // this->inNode->lock.lock();
```

```
    this->N++;
    // this->inNode->lock.unlock();
}
```

5.1.12.7.4 MCTS.cpp

```cpp
#include "MCTS.hpp"
#include <limits>
#include <jce/vector.hpp>

AlphaZero::ai::Node::Node(std::shared_ptr<Game::GameState> state
    )
{
  this->state = state;
}

AlphaZero::ai::Edge::Edge(Node* _outNode, Node* _inNode, int
    _action, float _p)
{
  this->P = _p;
  this->action = _action;
  this->outNode = _outNode;
  this->inNode = _inNode;
  this->N = 0;
  this->W = 0;
  this->Q = 0;
}

AlphaZero::ai::Edge::Edge()
{
  std::cout << "Edge default constructor" << std::endl;
  return;
}

std::pair<AlphaZero::ai::Node*, std::list<AlphaZero::ai::Edge*>>
    AlphaZero::ai::MCTS::moveToLeaf(Node* node)
{
  std::list<Edge*> backTrackList;
  while (true) {
    if (node->isLeaf()) {
      return { node, backTrackList };
    }
    else {
      float U;
      int Nb = 0;
      for (auto const& iter : node->edges)
      {
        Nb += iter.second.N;
      }
```

```cpp

      Edge* opsEdge = &(node->edges.begin()->second);
      int opsAction;

      float maxQu;
      bool nothasQU = true;

      for (auto& iter : node->edges) {
        U = U_computation(iter.second);
        if (nothasQU || U + iter.second.Q > maxQu) {
          opsEdge = &(iter.second);
          opsAction = iter.first;
          maxQu = U + iter.second.Q;
          nothasQU = false;
        }
      }
      opsEdge->traverse();
      backTrackList.push_back(opsEdge);
      node = opsEdge->outNode;
    }
  }
}

void AlphaZero::ai::MCTS::backFill(std::list<Edge*>& backTrace, Node* leaf, float val)
{
  float currentPlayer = (float)leaf->state->player * val;

  for (auto const& edge : backTrace) {
    // edge->inNode->lock.lock();

    edge->W = edge->W + currentPlayer * (float)edge->inNode->state->player;
    edge->Q = edge->W / (float)edge->N;

    // edge->inNode->lock.unlock();
  }
}
```

5.1.12.7.5 memory.hpp

```cpp
#pragma once
#include <iostream>
#include <fstream>
#include <sstream>

#include <queue>
#include <game/game.hpp>
#include "utils.hpp"
#include <jce/vector.hpp>


namespace AlphaZero {
  namespace ai {
      // class that represents a single Memory state
    class MemoryElement {
      // the MCTS computed value (value head)
    public: int value;

      // the game state the Memory element was built on.
    public: std::shared_ptr<Game::GameState>state;

      // the action values computed by the MCTS. (policy head)
    public: std::vector<float> av;

      // create a memory element around a game state and with certain action values.
    public: MemoryElement(std::shared_ptr<Game::GameState>, std::vector<int>);
      // create empty Element
    public: MemoryElement();
    };

      // holds the memory elemnt created by a game. The Elements are heald by the
      // temporary memory unitll the games winner is determined and the programm has
      // computed the value for every elemnt in tmp memory.
    class TemporaryMemory
    {
      // wether or not to write save elements (deactivate memory for tourney)
    public: bool active;

```

```
    // create temporary memory with active bool
    public: TemporaryMemory(bool);

    // vector used to save the Elements in.
    public: std::vector<std::shared_ptr<MemoryElement>>
tempMemory;

    // commit a game state and action value vector to memory
    // (Creata a Memory element and add it to the tempMemory
vector)
    public: void commit(std::shared_ptr<Game::GameState>, std::
vector<int>&);
    };

    // full memory used to train the model and receves data
from
    class Memory {
    // mutex used to ensure that only one tmp memory is read
in at any given time
    private: std::mutex mu;

    // wether or not the Memory should actualy save memory
Elements
    public: bool active = true;

    // build a temporary memory
    public: TemporaryMemory getTempMemory();

    // vector of all memory elements in the Memory
    public: std::vector<std::shared_ptr<MemoryElement>> memory;

    // constructor (dose nothing)
    public: Memory();

    // perform memory update from temporary memory and set
every elments value correctly
    public: void updateMemory(int player, int value,
TemporaryMemory* memory);

    // get a random state from the Memory
    public: std::shared_ptr<MemoryElement> getState();

    // save Memory to file with a certain version
    public: void save(int version);

```

```
      // call save(-1)
    public: void save();

      // save to a specific file name
    public: void save(char filename[]);

      // load specific memory version
    public: void load(int version);

      // call load(-1)
    public: void load();

      // load from custom file path
    public: void load(char filename[]);

      // render all states in memory (debuging)(depricated)
    public: void render();

      // subrutine of the public updateMemory function (dose all the actuall work)
    private: void updateMemory(int val, TemporaryMemory* memory);
    };
  }
}

inline AlphaZero::ai::TemporaryMemory::TemporaryMemory(bool val)
{
  this->active = val;
}

inline AlphaZero::ai::TemporaryMemory AlphaZero::ai::Memory::getTempMemory()
{
  AlphaZero::ai::TemporaryMemory memory(this->active);
  return memory;
}

inline AlphaZero::ai::MemoryElement::MemoryElement(std::shared_ptr<Game::GameState> _state, std::vector<int> _av)
{
  this->state = _state;
  float sum = (float)getSumm(_av);
  this->av = jce::vector::gen(_av.size(), 0.0f);
  for (size_t idx = 0; idx < this->av.size(); idx++)
```

```cpp
  {
    float tmp = ((float)_av[idx]);
    this->av[idx] = tmp / sum;
  }
}
inline AlphaZero::ai::MemoryElement::MemoryElement()
{
}
inline AlphaZero::ai::Memory::Memory()
{
}
inline void AlphaZero::ai::TemporaryMemory::commit(std::shared_ptr<Game::GameState> state, std::vector<int>& av)
{
  if (this->active) {
    std::vector<std::pair<std::shared_ptr<Game::GameState>, std::vector<int>>> idents = Game::identities(state, av);
    for (auto const& data : idents) {
      tempMemory.push_back(std::make_shared<MemoryElement>(data.first, data.second));
    }
  }
}

inline void AlphaZero::ai::Memory::updateMemory(int player, int value, TemporaryMemory* memory)
{
  this->updateMemory(player * value, memory);
}

inline std::shared_ptr<AlphaZero::ai::MemoryElement> AlphaZero::ai::Memory::getState()
{
  unsigned long long idx = rand() % this->memory.size();
  std::shared_ptr<MemoryElement> element = this->memory[idx];
  this->memory.erase(this->memory.begin() + idx);
  return element;
}

inline void AlphaZero::ai::Memory::render()
{
  for (auto const& element : this->memory)
  {
    element->state->render();
  }
```

```
}

inline void AlphaZero::ai::Memory::updateMemory(int val,
    TemporaryMemory* memory)
{
  while (memory->tempMemory.size() > 0) {
    std::shared_ptr<MemoryElement>& element = memory->tempMemory
    .back();
    element->value = element->state->player * val;
    this->mu.lock();
    this->memory.push_back(element);
    this->mu.unlock();
    memory->tempMemory.pop_back();
  }
}
```

5.1.12.7.6 memory.cpp

```cpp
#include "memory.hpp"
#include "config.hpp"
#include <jce/load.hpp>
#include <jce/save.hpp>

inline void getName(char out[], int version, int run)
{
  sprintf(out, "memory/run_%d/V_%d.memory", run, version);
}

void AlphaZero::ai::Memory::save(int version)
{
  char nameBuff[100];
  getName(nameBuff, version, runVersion);
  this->save(nameBuff);
}

void AlphaZero::ai::Memory::save()
{
  this->save(-1);
}

void AlphaZero::ai::Memory::save(char filename[])
{
  std::ofstream out(filename, std::ios::binary);
  if (out.is_open())
  {
    jce::save(out, this->memory);
    out.close();
  }
  else
  {
    throw "Game saver file not opend.";
  }
}

void AlphaZero::ai::Memory::load(int version)
{
  char nameBuff[100];
  getName(nameBuff, version, runVersion);
  this->load(nameBuff);
}

```

```
void AlphaZero::ai::Memory::load()
{
  char nameBuff[100];
  getName(nameBuff, -1, runVersion);
  try{
    std::cout << "Loading memory from file ... ";
    this->load(nameBuff);
    std::cout << "\33[1;32mSuccess\33[0m" << std::endl;
  }
  catch (...)
  {
    std::cout << "\33[1;31mFailed!\33[0m" << std::endl;
  }
}

void AlphaZero::ai::Memory::load(char filename[])
{
  std::ifstream in(filename, std::ios::binary);
  if (in.is_open())
  {
    jce::load(in, this->memory);
    in.close();
  }
  else
  {
    throw "Game saver file not opend.";
  }
}
```

5.1.12.7.7 model.hpp

```cpp
// TorchTestCMake.h : Include file for standard system include
    files,
// or project specific include files.

#pragma once

#include <iostream>
#include <game/game.hpp>
#include "memory.hpp"
#include <string>
#include <tuple>
#include <jce/string.hpp>
#include <string>
#include <cmath>
#include "modelWorker.hpp"
#include <jce/save.hpp>
#include <jce/load.hpp>
#include <test/testUtils.hpp>


namespace AlphaZero {
  namespace ai {
    // class that implements the top layer wich consists of the
    following sub-layers:
    //  - Conv2d with kernel size of 3x3
    //  - LayerNorm
    //  - LeakyReLu
    class TopLayer : public torch::nn::Module {
    public: torch::nn::Conv2d conv1;
    public: torch::nn::LayerNorm batch;
    public: torch::nn::LeakyReLU relu;
    private: int kernel1;

    public: TopLayer(int inp, int out, int kernelsize1);
    public: torch::Tensor forward(torch::Tensor);
    public: void moveTo(c10::Device device);
    };
    // Residual layer consisting of the following sub-layers:
    //   - Conv2d with a 3x3 kernel
    //   - LayerNorm
    //   - LeakyReLU
    class ResNet : public torch::nn::Module {
    public: torch::nn::Conv2d conv1, conv2;
```

```
public: torch::nn::LayerNorm batch, batch2;
public: torch::nn::LeakyReLU activ;
private: int kernel1, kernel2;

public: ResNet(int inp, int out, int kernelsize1, int kernelsize2);
public: torch::Tensor forward(torch::Tensor);
public: void moveTo(c10::Device device);
};

// value head layer that is used to compute the value of a game state. It consists of the following sub-layers:
//   - Conv2D with kernel size 1x1
//   - LeakyReLU
//   - flatten
//   - Linear layer (lin1)
//   - LeakyRelu
//   - Linear Layer (lin2)
//   - tanh
class Value_head : torch::nn::Module {
private: bool isSecondRun = false;
private: torch::Tensor tmpX;

public: torch::nn::Conv2d conv;
public: torch::nn::Linear lin1, lin2;
public: torch::nn::LeakyReLU relu;
public: torch::nn::Tanh tanh;
private: int size;

public: Value_head(int inp, int hidden_size, int out, int kernels);
public: torch::Tensor forward(torch::Tensor);
public: void moveTo(c10::Device device);
};

// policy head used to compute the policy of a certain state. It consists of the following sub-layers
//   - Conv2d with a kernel size of 1x1
//   - LeakyReLU
//   - flatten
//   - Linear layer
class Policy_head : torch::nn::Module {
public: torch::nn::Conv2d conv;
public: torch::nn::Linear lin1;
public: torch::nn::LeakyReLU relu;
```

```
private: int size;

public: Policy_head(int inp, int hidden, int out);
public: torch::Tensor forward(torch::Tensor);
public: void moveTo(c10::Device device);
};

  // some type definitions [...]
typedef torch::nn::MSELoss Loss;
typedef torch::optim::SGD Optimizer;
typedef torch::optim::SGDOptions OptimizerOptions;

// full model consisting of the following layers:
//   - TopLayer (75 fillters)
//   - 6 ResidualBlocks (75 fillters)
//    *** the model splits into two heads (ValueHead and
PolicyHead**
//   - ValueHead (10 fillters, lin1 size: 210, lin2 size: 1)

//   - policyHead (2 fillters, linear size: 42)
class Model : public torch::nn::Module {
  //private: torch::nn::Conv2d headLayer;
private: TopLayer top;
private: ResNet res1, res2, res3, res4, res5, res6;
private: Value_head value_head;
private: Policy_head policy_head;

private: char* device;

private: Loss loss;
private: Optimizer optim;

public: Model(char* device);
public: std::pair<torch::Tensor, torch::Tensor> forward(
torch::Tensor);

  // train the nework to output a certain result
public: std::pair<float, float> train(const std::pair<torch
::Tensor, torch::Tensor>& x, const std::pair<torch::Tensor,
torch::Tensor>& y);

  // pass game state though model (encode to tensor and pass
 forward)
public: std::pair<float, torch::Tensor>predict(std::
shared_ptr<Game::GameState> state);
```

```cpp

        // get data batch from memory
    public: static std::tuple<torch::Tensor, torch::Tensor, torch::Tensor> getBatch(std::shared_ptr<Memory> memory, unsigned int batchSize);

      // forward pass with model data
    public: void predict(ModelData* data);

      // forward pass with multiple model data containers
    public: void predict(std::list<ModelData*> data);

      // forward though network than call train
    public: void fit(const std::tuple<torch::Tensor, torch::Tensor, torch::Tensor>& batch, const unsigned short& run, const unsigned short& trainingLoop);

      // save version of model
    public: void save_version(unsigned int version);

      // save a current version
    public: void save_as_current();

      // save to a specific file
    public: void save_to_file(char* filename);
      // alternative jce save
    public: void jce_save_current(char* filename);

      // load specific model version
    public: void load_version(unsigned int version);

      // load current model
    public: void load_current();

      // load model from given file
    public: void load_from_file(char* filename);

      // alternative jce loading method
    public: void jce_load_from_file(char* filename);

      // copy model into this one
    public: void copyModel(Model*);
    private: void copyParameters(torch::OrderedDict<std::string, torch::Tensor> prams);

```

```cpp
      // move model to diferant device
    public: void moveTo(c10::Device device);


      // functions used to add custom modules to this graph.
    private: TopLayer register_custom_module(TopLayer net);
    private: ResNet register_custom_module(ResNet net, std::string layer);
    private: Value_head register_custom_module(Value_head net);
    private: Policy_head register_custom_module(Policy_head net);

    };
  }
}
// customizable section
#define modelTest false
#define randomModel false
#define convSize 5

inline AlphaZero::ai::Model::Model(char* _device) :
  top(this->register_custom_module(TopLayer(2, 75, convSize))),
  res1(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_1")),
  res2(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_2")),
  res3(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_3")),
  res4(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_4")),
  res5(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_5")),
  res6(this->register_custom_module(ResNet(75, 75, convSize, convSize), "Residual_6")),
  value_head(this->register_custom_module(Value_head(75, 420, 210, 10))),
  policy_head(this->register_custom_module(Policy_head(75, 84, 42))),
  optim(Optimizer(this->parameters(), OptimizerOptions(learningRage).momentum(Momentum))),
  device(_device)
{
  this->moveTo(c10::Device(_device));
}

```

```cpp
inline std::pair<torch::Tensor, torch::Tensor> AlphaZero::ai::Model::forward(torch::Tensor x)
{
#if randomModel
  return { torch::rand({x.size(0), 1}), torch::rand({x.size(0), action_count}) };
#else
  x = this->top.forward(x);
  x = this->res1.forward(x);
  x = this->res2.forward(x);
  x = this->res3.forward(x);
  x = this->res4.forward(x);
  x = this->res5.forward(x);
  x = this->res6.forward(x);

  // compute individual heads
  torch::Tensor value = this->value_head.forward(x.clone());
  torch::Tensor poly = this->policy_head.forward(x.clone());

  return { value, poly };
#endif
};
// end of cutimizable section

inline AlphaZero::ai::TopLayer::TopLayer(int inp, int out, int kernelsize1) :
  conv1(this->register_module("conv1", torch::nn::Conv2d(torch::nn::Conv2dOptions(inp, out, kernelsize1)))),
  batch(this->register_module("batch", torch::nn::LayerNorm(torch::nn::LayerNormOptions({ out, input_shape_y, input_shape_x })))),
  relu(this->register_module("ReLU", torch::nn::LeakyReLU(torch::nn::LeakyReLU()))),
  kernel1(kernelsize1 / 2)
{
}
inline torch::Tensor AlphaZero::ai::TopLayer::forward(torch::Tensor x)
{
  x = torch::nn::functional::pad(x, torch::nn::functional::PadFuncOptions({ kernel1, kernel1, kernel1, kernel1 }));
  x = this->conv1(x);
  x = this->batch(x);
  x = this->relu(x);
  return x;
```

```
}

inline void AlphaZero::ai::TopLayer::moveTo(c10::Device device)
{
  this->conv1->to(device, true);
  this->batch->to(device, true);
  this->relu->to(device, true);
}

inline AlphaZero::ai::ResNet::ResNet(int inp, int out, int kernelsize1, int kernelsize2) :
  kernel1(kernelsize1), kernel2(kernelsize2),
  conv1(this->register_module("conv1", torch::nn::Conv2d(torch::nn::Conv2dOptions(inp, out, kernelsize1)))),
  conv2(this->register_module("conv2", torch::nn::Conv2d(torch::nn::Conv2dOptions(out, out, kernelsize2)))),
  batch(this->register_module("batch1", torch::nn::LayerNorm(torch::nn::LayerNormOptions({ out, input_shape_y, input_shape_x })))),
  batch2(this->register_module("batch2", torch::nn::LayerNorm(torch::nn::LayerNormOptions({ out, input_shape_y, input_shape_x })))),
  activ(this->register_module("activ", torch::nn::LeakyReLU(torch::nn::LeakyReLU())))
{
  if (torch::cuda::is_available()) {
    this->moveTo(c10::Device("cuda:0"));
  }
}

inline torch::Tensor AlphaZero::ai::ResNet::forward(torch::Tensor x)
{
#if modelTest
  std::cout << x.sizes() << std::endl;
#endif
  auto y = x.clone();
  x = torch::nn::functional::pad(x, torch::nn::functional::PadFuncOptions({ kernel1 / 2, kernel1 / 2, kernel1 / 2, kernel1 / 2 }));
  x = this->conv1(x);
  x = this->batch(x);
  x = this->activ(x);

  x = torch::nn::functional::pad(x, torch::nn::functional::
```

```
    PadFuncOptions({ kernel2 / 2, kernel2 / 2, kernel2 / 2,
    kernel2 / 2 }));
  x = this->conv2(x);
  x = this->batch2(x);
  return this->activ(x + y);
}

inline void AlphaZero::ai::ResNet::moveTo(c10::Device device)
{
  this->conv1->to(device, true);
  this->conv2->to(device, true);
  this->batch->to(device, true);
  this->batch2->to(device, true);
  this->activ->to(device, true);
}

inline AlphaZero::ai::Value_head::Value_head(int inp, int
    hidden_size, int out, int convOut) :
  conv(this->register_module("conv", torch::nn::Conv2d(torch::nn
    ::Conv2dOptions(inp, convOut, 1)))),
  lin1(this->register_module("lin1", torch::nn::Linear(torch::nn
    ::LinearOptions(hidden_size, out)))),
  lin2(this->register_module("lin2", torch::nn::Linear(torch::nn
    ::LinearOptions(out, 1)))),
  relu(this->register_module("relu", torch::nn::LeakyReLU())),
  tanh(this->register_module("tanh", torch::nn::Tanh()))
{
  this->size = hidden_size;

  if (torch::cuda::is_available())
  {
    this->moveTo(c10::Device("cuda:0"));
  }
}

inline torch::Tensor AlphaZero::ai::Value_head::forward(torch::
    Tensor x)
{
#if modelTest
  std::cout << "value" << std::endl;
  std::cout << x.sizes() << std::endl;
#endif
  x = this->conv(x);
  x = this->relu(x);
  x = this->lin1(x.reshape({ x.size(0), this->size }));
```

```cpp
  x = this->relu(x);
  x = this->lin2(x);
  x = this->tanh(x);
  return x;
}

inline void AlphaZero::ai::Value_head::moveTo(c10::Device device
    )
{
  this->conv->to(device, true);
  this->lin1->to(device, true);
  this->relu->to(device, true);
  this->lin2->to(device, true);
  this->tanh->to(device, true);
}

inline AlphaZero::ai::Policy_head::Policy_head(int inp, int
    hidden, int out) :
  conv(this->register_module("conv", torch::nn::Conv2d(torch::nn
    ::Conv2dOptions(inp, 2, 1)))),
  lin1(this->register_module("lin1", torch::nn::Linear(hidden,
    out))),
  relu(this->register_module("relu", torch::nn::LeakyReLU()))
{
  this->size = hidden;

  if (torch::cuda::is_available()) {
    this->moveTo(c10::Device("cuda:0"));
  }
}

inline torch::Tensor AlphaZero::ai::Policy_head::forward(torch::
    Tensor x)
{
#if modelTest
  std::cout << "poly" << std::endl;
  std::cout << x.sizes() << std::endl;
#endif
  x = this->conv(x);
  x = this->relu(x);
  x = this->lin1(x.reshape({ x.size(0), this->size }));
  return x;
}

inline void AlphaZero::ai::Policy_head::moveTo(c10::Device
```

```
    device)
{
  this->conv->to(device, true);
  this->lin1->to(device, true);
}

inline torch::Tensor polyLoss(torch::Tensor a, torch::Tensor b)
{
#if true
  auto c = torch::where(b == 0, a, b);
  return torch::mse_loss(a, c);
#else
  return torch::mse_loss(a, b);
#endif
}

inline std::pair<float, float> AlphaZero::ai::Model::train(const
     std::pair<torch::Tensor, torch::Tensor>& x, const std::pair<
    torch::Tensor, torch::Tensor>& y)
{
  //std::cout << x.first << std::endl << y.first << std::endl;

  auto valLoss = 0.5f * torch::mse_loss(x.first, y.first);
  auto plyLoss = 0.5f * polyLoss(x.second, y.second);
  auto loss = (valLoss + plyLoss);

  loss.backward();
  this->optim.step();
  this->optim.zero_grad();
  std::pair<float, float> error = { torch::mean(valLoss).item().
    toFloat(), torch::mean(plyLoss).item().toFloat()};

  if (std::isnan(error.first))
  {
    //std::cout << valLoss << std::endl << plyLoss << std::endl;
    std::cout << x.first  << std::endl << y.first  << std::endl;
    std::cout << x.second << std::endl << y.second << std::endl;
    return error;
  }

  return error;
}

inline std::pair<float, torch::Tensor> AlphaZero::ai::Model::
    predict(std::shared_ptr<Game::GameState> state)
```

```
{
  torch::Tensor NNInput = state->toTensor().to(c10::Device(this->device));
  std::pair<torch::Tensor, torch::Tensor> NNOut = this->forward(NNInput);
  float value = NNOut.first[0].item<float>();
  return { value, NNOut.second };
}

inline void AlphaZero::ai::Model::predict(ModelData* data)
{
  torch::Tensor NNInput = data->node->state->toTensor().to(c10::Device(this->device));
  std::pair<torch::Tensor, torch::Tensor> NNOut = this->forward(NNInput);

  torch::Tensor mask = torch::ones(
    { 1, action_count },
    c10::TensorOptions().device(c10::Device("cpu")).dtype(at::kBool)
  );

  for (auto idx : data->node->state->allowedActions)
  {
    mask[0][idx] = false;
  }
  //std::cout << std::endl << NNOut.first << std::endl << NNOut.second << std::endl;
  data->value = NNOut.first[0].item<float>();
  data->polys = torch::softmax(torch::masked_fill(NNOut.second.cpu(), mask, -1000.0f), 1)[0];
}

inline void AlphaZero::ai::Model::predict(std::list<ModelData*> data)
{
  torch::Tensor NNInput = torch::zeros({ (int)data.size(), input_snape_z, input_shape_y, input_shape_x });
  torch::Tensor mask = torch::ones(
    { (int)data.size(), action_count },
    c10::TensorOptions().device(c10::Device("cpu")).dtype(at::kBool)
  );
  auto iter = data.begin();
  for (unsigned short idx = 0; idx < data.size(); idx++)
```

```
  {
    NNInput[idx] = (*iter)->node->state->toTensor()[0];
    for (auto action : (*iter)->node->state->allowedActions)
    {
      mask[idx][action] = false;
    }
    iter++;
  }

  std::pair<torch::Tensor, torch::Tensor> NNOut = this->forward(NNInput.to(c10::Device(this->device)));

  //std::cout << std::endl << NNOut.first << std::endl << NNOut.second << std::endl;
  mask = mask.to(c10::Device(this->device));

  auto soft = torch::softmax(torch::masked_fill(NNOut.second, mask, -1000.0f), 1).cpu();

  iter = data.begin();
  for (unsigned int idx = 0; idx < data.size(); idx++)
  {
    (*iter)->value = NNOut.first[idx].item<float>();
    (*iter)->polys = soft[idx];
    iter++;
  }
}

inline std::tuple<torch::Tensor, torch::Tensor, torch::Tensor> AlphaZero::ai::Model::getBatch(std::shared_ptr<Memory> memory, unsigned int batchSize)
{
  std::tuple<torch::Tensor, torch::Tensor, torch::Tensor> output =
  {
    at::zeros({batchSize, input_snape_z, input_shape_y, input_shape_x}),
    at::zeros({batchSize, action_count}),
    at::zeros({batchSize, 1})
  };
  for (unsigned short idx = 0; idx < batchSize; idx++) {
    auto state = memory->getState();
    state->state->toTensor(std::get<0>(output), idx);
    std::get<1>(output)[idx] = at::from_blob(state->av.data(), { action_count }).toType(torch::kFloat16);
```

```cpp
    std::get<2>(output)[idx] = torch::tensor({ state->value });
  }
  return output;
}

inline void AlphaZero::ai::Model::fit(const std::tuple<torch::Tensor, torch::Tensor, torch::Tensor>& batch, const unsigned short& run, const unsigned short& trainingLoop)
{
  std::pair<torch::Tensor, torch::Tensor> NNVals = this->forward(std::get<0>(batch).to(c10::Device(this->device)));
  std::pair<float, float> error = this->train(NNVals,
    {
    std::get<2>(batch).to(c10::Device(this->device)),
    std::get<1>(batch).to(c10::Device(this->device))
    });
#if ModelLogger
  debug::log::modelLogger->info("model error in iteration {} on batch {} had valueError of {} and polyError of {}", run, trainingLoop, std::get<0>(error), std::get<1>(error));
#endif
#if LossLogger
  debug::log::_lossLogger.addValue(error);
#endif
}

inline void AlphaZero::ai::Model::save_version(unsigned int version)
{
  char buffer[50];
  std::sprintf(buffer, "models/run_%d/V_%d.torch", runVersion, version);
  std::cout << buffer << std::endl;
  this->save_to_file(buffer);
}

inline void AlphaZero::ai::Model::save_as_current()
{
  char buffer[50];
  std::sprintf(buffer, "models/run_%d/currentModel.torch", runVersion);
  this->save_to_file(buffer);
}

inline void AlphaZero::ai::Model::save_to_file(char* filename)
```

```
{
  torch::serialize::OutputArchive out;
  this->save(out);
  std::string model_path = std::string(filename);
  out.save_to(model_path);
}

inline void AlphaZero::ai::Model::jce_save_current(char*
    filename)
{
  std::ofstream out(filename, std::ios::binary);
  jce::save(out, this->named_parameters(true));
  out.close();
}


inline void AlphaZero::ai::Model::load_version(unsigned int
    version)
{
  std::cout << "loading ...";
  char buffer[50];
  std::sprintf(buffer, "models/run_%d/V_%d.torch", runVersion,
    version);
  this->load_from_file(buffer);
  std::cout << " loaded Version " << version << std::endl;
}

inline void AlphaZero::ai::Model::load_current()
{
  char buffer[50];
  std::sprintf(buffer, "models/run_%d/currentModel.torch",
    runVersion);
  this->load_from_file(buffer);
}

inline void AlphaZero::ai::Model::load_from_file(char* filename)
{
  torch::serialize::InputArchive inp;
  std::string model_path = std::string(filename);
  inp.load_from(model_path);
  this->load(inp);
}

inline void AlphaZero::ai::Model::jce_load_from_file(char*
    filename)
```

```cpp
{
  std::cout << "loading ...\t";
  torch::autograd::GradMode::set_enabled(false);
  torch::OrderedDict<std::string, torch::Tensor> map;
  std::ifstream in(filename, std::ios::binary);
  if (in.is_open())
  {
    jce::load(in, map);
    this->copyParameters(map);
  }
  test::printSuccess(in.is_open());
  in.close();
  torch::autograd::GradMode::set_enabled(true);
}

inline void AlphaZero::ai::Model::copyModel(AlphaZero::ai::Model* model)
{
  torch::autograd::GradMode::set_enabled(false);
  auto new_params = model->named_parameters(true);
  this->copyParameters(new_params);
  torch::autograd::GradMode::set_enabled(true);
}

inline void AlphaZero::ai::Model::copyParameters(torch::OrderedDict<std::string, torch::Tensor> new_params)
{
  auto params = this->named_parameters(true);
  auto buffers = this->named_buffers(true);
  for (auto& val : new_params) {
    auto name = val.key();
    auto* t = params.find(name);
    if (t != nullptr) {
      t->copy_(val.value());
    }
    else {
      t = buffers.find(name);
      if (t != nullptr) {
        t->copy_(val.value());
      }
    }
  }
}

inline void AlphaZero::ai::Model::moveTo(c10::Device device)
```

```
{
  this->top.moveTo(device);

  this->res1.moveTo(device);
  this->res2.moveTo(device);
  this->res3.moveTo(device);
  this->res4.moveTo(device);
  this->res5.moveTo(device);
  this->res6.moveTo(device);

  this->value_head.moveTo(device);
  this->policy_head.moveTo(device);
}

inline AlphaZero::ai::TopLayer AlphaZero::ai::Model::
    register_custom_module(TopLayer net)
{
  register_module("TopLayer_conv", net.conv1);
  register_module("TopLayer_batch", net.batch);
  register_module("TopLayer_ReLU", net.relu);
  return net;
}

inline AlphaZero::ai::ResNet AlphaZero::ai::Model::
    register_custom_module(ResNet net, std::string layer)
{
  register_module(layer + "_conv1", net.conv1);
  register_module(layer + "_conv2", net.conv2);
  register_module(layer + "_batch1", net.batch);
  register_module(layer + "_batch2", net.batch2);
  register_module(layer + "_active", net.activ);
  return net;
}
inline AlphaZero::ai::Value_head AlphaZero::ai::Model::
    register_custom_module(Value_head net)
{
  register_module("value_conv", net.conv);
  register_module("value_lin1", net.lin1);
  register_module("value_lin2", net.lin2);
  register_module("value_ReLU", net.relu);
  register_module("value_tanh", net.tanh);
  return net;
}
inline AlphaZero::ai::Policy_head AlphaZero::ai::Model::
    register_custom_module(Policy_head net)
```

```
{
  register_module("policy_conv", net.conv);
  register_module("policy_linear", net.lin1);
  return net;
}
```

5.1.12.7.8 modelSynchronizer.hpp

```cpp
#pragma once

#include <mutex>
#include <thread>
#include <thread>
#include <iostream>
#include <memory>
#include "model.hpp"

// this is compleatly useless is it not.

namespace AlphaZero
{
  namespace ai
  {
    // interface betwean model.hpp and agent.hpp. It handles synchronization of models over multiple devices.
    class ModelSynchronizer
    {

      // array containing the actual models
    private: std::vector<std::unique_ptr<Model>> models;

      // model index iteratror.
    private: unsigned short pos = 0;

      // mutex used to split the threads over the devices.
    private: std::mutex modelGetMutex;

      // create models on devices and generate synchronizer.
    public: ModelSynchronizer(std::vector<char*> devices);

      // evaluate Model data
    public: void addData(ModelData* data);

      // get the model to perform the prediction
    private: Model* getModel();

      // copy model[0] from other synchronizer and synchronize its models.
    public: void copyModel(ModelSynchronizer*);

      // trains models[0] and updatas the others the same way.
```

```
    public: void fit(const std::tuple<torch::Tensor, torch::Tensor, torch::Tensor>& batch, const unsigned short& run, const unsigned short& trainingLoop);

      // save models[0] as current
    public: void save_as_current();

      // save models[0] under a certain version
    public: void save_version(unsigned int version);

      // save models[0] to a specific file
    public: void save_to_file(char* filename);

      // save models[0] as jce format
    public: void jce_save_current(char* filename);

      // load current model file and call synchronizeModels
    public: void load_current();

      // load model by version from file and call synchronizeModels
    public: void load_version(unsigned int version);

      // load model from custom file file and call synchronizeModels
    public: void load_from_file(char* filename);

      // load model file in jce format and call synchronizeModels
    public: void jce_load_from_file(char* filename);

      // sets the parameters of all models to that of models[0]
    public: void synchronizeModels();

      // generates the model data and calls addData for evaluation
    public: std::pair<float, torch::Tensor>predict(std::shared_ptr<Game::GameState> state, size_t idx=0);
    public: void predict(ModelData* data, size_t idx=0);
    public: void predict(std::list<ModelData*> data, size_t idx=0);
    };
}
namespace test
{
```

```
    namespace ModelSynchronizer
    {
      std::thread addTestData(ai::ModelData* data, ai::ModelSynchronizer* sync);
      void _addTestData(ai::ModelData* data, ai::ModelSynchronizer* sync);
    }
  }
}

inline AlphaZero::ai::ModelSynchronizer::ModelSynchronizer(std::vector<char*> devices)
{
  for (auto const& device : devices)
  {
    this->models.push_back(std::make_unique<Model>(device));
  }
  this->synchronizeModels();
}
inline void AlphaZero::ai::ModelSynchronizer::addData(ModelData* _data)
{
  /*_data->value = 2;
  std::list<ModelData*>data_l;
  data_l.push_back(_data);*/
  this->getModel()->predict(_data);
}
inline AlphaZero::ai::Model* AlphaZero::ai::ModelSynchronizer::getModel()
{
  this->modelGetMutex.lock();
  auto outputModel = this->models[this->pos].get();
  this->pos++;
  if (this->pos >= this->models.size())
  {
    this->pos = 0;
  }
  this->modelGetMutex.unlock();
  return outputModel;
}
inline void AlphaZero::ai::ModelSynchronizer::copyModel(ModelSynchronizer* syncher)
{
  for (auto const& model : this->models)
  {
```

```
    model->copyModel(syncher->models[0].get());
  }
}
inline void AlphaZero::ai::ModelSynchronizer::fit(const std::tuple<torch::Tensor, torch::Tensor, torch::Tensor>& batch, const unsigned short& run, const unsigned short& trainingLoop)
{
  this->models[0]->fit(batch, run, trainingLoop);
}
inline void AlphaZero::ai::ModelSynchronizer::save_as_current()
{
  this->models[0]->save_as_current();
}
inline void AlphaZero::ai::ModelSynchronizer::save_to_file(char* filename)
{
  this->models[0]->save_to_file(filename);
}
inline void AlphaZero::ai::ModelSynchronizer::jce_save_current(char* filename)
{
  this->models[0]->jce_save_current(filename);
}
inline void AlphaZero::ai::ModelSynchronizer::save_version(unsigned int version)
{
  this->models[0]->save_version(version);
}
inline void AlphaZero::ai::ModelSynchronizer::load_current()
{
  this->models[0]->load_current();
  this->synchronizeModels();
}
inline void AlphaZero::ai::ModelSynchronizer::load_from_file(char* filename)
{
  this->models[0]->load_from_file(filename);
  this->synchronizeModels();
}
inline void AlphaZero::ai::ModelSynchronizer::jce_load_from_file(char* filename)
{
  this->models[0]->jce_load_from_file(filename);
}
```

```cpp
inline void AlphaZero::ai::ModelSynchronizer::load_version(unsigned int version)
{
  this->models[0]->load_version(version);
  this->synchronizeModels();
}
inline void AlphaZero::ai::ModelSynchronizer::synchronizeModels()
{
  auto copyFrom = this->models[0].get();
  for (auto const& model : this->models)
  {
    if (model.get() != copyFrom)
    {
      model->copyModel(copyFrom);
    }
  }
}
inline void AlphaZero::ai::ModelSynchronizer::predict(ModelData* data, size_t idx)
{
  this->models[idx]->predict(data);
}
inline void AlphaZero::ai::ModelSynchronizer::predict(std::list<ModelData*> data, size_t idx)
{
  this->models[idx]->predict(data);
}
inline std::pair<float, torch::Tensor> AlphaZero::ai::ModelSynchronizer::predict(std::shared_ptr<Game::GameState> state, size_t idx)
{
  return this->models[idx]->predict(state);
}


inline std::thread AlphaZero::test::ModelSynchronizer::addTestData(ai::ModelData* data, ai::ModelSynchronizer* sync)
{
  std::thread thread(_addTestData, data, sync);
  return thread;
}

inline void AlphaZero::test::ModelSynchronizer::_addTestData(ai::ModelData* data, ai::ModelSynchronizer* sync)
```

```
{
  sync->addData(data);
}
```

5.1.12.7.9 modelWorker.hpp

```cpp
#pragma once

#include "MCTS.hpp"

// also useless

namespace AlphaZero
{
  namespace ai
  {
    class Node;

    class ModelData
    {
    public: Node* node;
    public: torch::Tensor polys;
    public: float value;

    public: ModelData(Node* node);
    public: void print();
    };
  }
}

inline AlphaZero::ai::ModelData::ModelData(Node* _node)
{
  this->node = _node;
}

inline void print(){}
```

5.1.12.7.10 modelWorker.cpp

```
#include "modelWorker.hpp"
#include "MCTS.hpp"

/*
void AlphaZero::ai::ModelData::print()
{
  std::cout << "state: " << std::endl;
  this->node->state->render();
  std::cout << "polys: " << std::endl;
  std::cout << torch::reshape(this->polys, { action_shape }) <<
    std::endl;
  std::cout << "value: " << this->value << std::endl;
}*/
```

5.1.12.7.11 playGame.hpp

```cpp
#pragma once

#include <ai/agent.hpp>
#include <ai/memory.hpp>


namespace AlphaZero {
  namespace ai {
    struct gameOutput
    {

        // map that counts how often a player has won.
      std::unordered_map<Agent*, int> map;
        // mutex used updating 'map' from multiple threads
      std::mutex ex;

        // create a gameOutput object for games betwean tow agents
      gameOutput(Agent*, Agent*);
        // add a win for a certain player.
      void updateValue(Agent*);
    };
      // main training loop
      // create agents, generate data though self play, train an agent, and evaluate the next one. (save if better)
    void train(int);
      // play a certain amount of games between two (can be the same one) and record the actionValues and states.
    void playGames(gameOutput* output, Agent* agent1, Agent* agent2, Memory* memory, int probMoves, int Epochs, char RunId[], int goesFist = 0, bool log = false);
      // create the various threads the games run in and collect the data at the end.
    std::unordered_map<Agent*, int> playGames_inThreads(Game::Game* game, Agent* agent1, Agent* agend2, Memory* memory, int probMoves, int Epochs, int Threads, char RunId[], int goesFirst = 0, bool log = false);
  }
  namespace test {
      // play test game
    void playGame(std::shared_ptr<Game::Game> game, std::shared_ptr<ai::Agent> player1, std::shared_ptr<ai::Agent> player2, int goesFirst=0);
```

```
  }
}

inline std::unordered_map<AlphaZero::ai::Agent*, int> AlphaZero
    ::ai::playGames_inThreads(Game::Game* game, Agent* agent1,
    Agent* agent2, Memory* memory, int probMoves, int Epochs, int
     Threads, char RunId[], int goesFirst, bool log)
{
  gameOutput output(agent1, agent2);

  std::vector<std::thread> workers;
  for (size_t idx = 0; idx < Threads; idx++)
  {
    bool doLog = (log && (idx == 0));
    workers.push_back(std::thread(playGames, &output, agent1,
    agent2, memory, probMoves, Epochs, RunId, goesFirst, doLog));
  }

  for (auto& worker : workers)
  {
    worker.join();
  }

  return output.map;
}

inline void AlphaZero::ai::gameOutput::updateValue(Agent* idx)
{
  this->ex.lock();
  this->map[idx] = this->map[idx] + 1;
  this->ex.unlock();
}

inline AlphaZero::ai::gameOutput::gameOutput(Agent* agent1,
    Agent* agent2)
{
  this->map.insert({ agent1, 0 });
  this->map.insert({ agent2, 0 });
}
```

5.1.12.7.12 playGame.cpp

```cpp
#include "playGame.hpp"
#include <jce/load.hpp>
#include <jce/save.hpp>
#include <io.hpp>


void AlphaZero::test::playGame(std::shared_ptr<Game::Game> game,
    std::shared_ptr<ai::Agent> player1, std::shared_ptr<ai::
    Agent> player2, int goesFirst)
{
  if (goesFirst == 0) {
    goesFirst = 1;
    if (rand() % 2) {
      goesFirst = -1;
    }
  }
  player1->reset();
  player2->reset();
  std::unordered_map<int, std::shared_ptr<ai::Agent>> players =
    { {goesFirst, player1},{-goesFirst, player2} };
  int action;
  while (!game->state->done) {
    action = players[game->state->player]->getAction(game->state
    , false).first;
    game->takeAction(action);
  }
}

void AlphaZero::ai::train(int version)
{
  unsigned short iteration = 0;
  std::vector<char*> devices = { DEVICES };
  std::shared_ptr<Memory> memory = std::make_shared<Memory>();
  std::shared_ptr<Game::Game> game = std::make_shared<Game::Game
    >();
  std::shared_ptr<Agent> currentAgent = std::make_shared<Agent>(
    devices);
  std::shared_ptr<Agent> bestAgent = std::make_shared<Agent>(
    devices);

  std::vector<int> requiredIterations;

  memory->load();
```

```
  char nameBuff[100];

  std::sprintf(nameBuff, "models/run_%d/versionCount.jce", runVersion);
  std::ifstream fin(nameBuff, std::ios::binary);
  if (fin.is_open())
  {
    jce::load(fin, version);
    std::cout << "found model version: " << version << std::endl;

    fin.close();

    std::sprintf(nameBuff, "models/run_%d/iterationCounter.jce", runVersion);
    fin.open(nameBuff, std::ios::binary);
    if (fin.is_open())
    {
      jce::load(fin, requiredIterations);
      std::cout << "loaded required Iterations: it hs size: " << requiredIterations.size() << std::endl;
    }
    else
    {
      std::cout << "could not find sutalbe iterationCounter";
    }
    fin.close();

    bestAgent->model->load_version(version);
    currentAgent->model->load_version(version);
  }
  else
  {
    std::cout << "model version config not found. Defaulting to 0" << std::endl;
    version = 0;
  }
  currentAgent->model->copyModel(bestAgent->model.get());

  while (true) {
    iteration++;
    memory->active = true;
#if MainLogger
    debug::log::mainLogger->info("playing version: {}", version);
```

```cpp
#endif

    std::cout << "playing Generational Games:" << std::endl;

#if ModelLogger
    debug::log::modelLogger->info("Running Training Games");
#endif

    sprintf(nameBuff, "logs/games/game_%d_Generator.gameLog", iteration);
    playGames_inThreads(game.get(), bestAgent.get(), bestAgent.get(), memory.get(), probabilitic_moves, EPOCHS, GEN_THREADS, nameBuff, 1, false);
    std::cout << "memory size is: " << memory->memory.size() << std::endl;
    if (memory->memory.size() > memory_size) {
#if ProfileLogger
      debug::Profiler::profiler.switchOperation(5);
#endif
      currentAgent->fit(memory, iteration);
#if ProfileLogger
      debug::Profiler::profiler.stop();
#endif
      memory->active = false;
      std::cout << "playing Tournement Games:" << std::endl;
#if MainLogger
      debug::log::mainLogger->info("RETRAINING ==================================================");
#endif

      sprintf(nameBuff, "logs/games/game_%d_Turney.gameLog", iteration);
#if ModelLogger
      debug::log::modelLogger->info("Running Tourney Games");
#endif
      auto score = playGames_inThreads(game.get(), bestAgent.get(), currentAgent.get(), memory.get(), Turnement_probabiliticMoves, TurneyEpochs, TurneyThreads, nameBuff, 0, true);

      std::cout << "Turney ended with: " << score[currentAgent.get()] << " : " << score[bestAgent.get()] << std::endl;
      if (score[currentAgent.get()] > score[bestAgent.get()] * scoringThreshold) {
        version++;
```

```
        currentAgent->model->save_as_current();
        bestAgent->model->copyModel(currentAgent->model.get());
        bestAgent->model->save_version(version);

        std::sprintf(nameBuff, "models/run_%d/versionCount.jce", runVersion);
        std::ofstream fout(nameBuff, std::ios::binary);
        jce::save(fout, version);
        fout.close();

        memory->save();

        requiredIterations.push_back(iteration);
        std::sprintf(nameBuff, "models/run_%d/iterationCounter.jce", runVersion);
        fout.open(nameBuff, std::ios::binary);
        jce::save(fout, requiredIterations);
        fout.close();

        iteration = 0;
      }
    }
  }
}

void AlphaZero::ai::playGames(gameOutput* output, Agent* agent1, Agent* agent2, Memory* memory, int probMoves, int Epochs, char RunId[], int _goesFist, bool do_log)
{
  std::srand(std::chrono::time_point_cast<std::chrono::nanoseconds>(std::chrono::system_clock::now()).time_since_epoch().count());
  auto game = std::make_unique<Game::Game>();
  int goesFist = (_goesFist == 0) ? 1 : _goesFist;
#if ProfileLogger
  debug::Profiler::profiler.switchOperation(3);
#endif

#if SaverType == 1
  io::FullState::GameSaver saver = io::FullState::GameSaver();
#elif SaverType == 2
  io::ActionsOnly::GameSaver saver = io::ActionsOnly::GameSaver();
#endif
```

```

  for (int epoch = 0; epoch < Epochs; epoch++) {
#if RenderGenAndTurneyProgress
    jce::consoleUtils::render_progress_bar((float)epoch / (float)Epochs);
#endif
#if ProfileLogger
    debug::Profiler::profiler.switchOperation(3);
#endif
#if SaverType == 2 || SaverType == 1
    saver.addGame();
#endif
#if MainLogger
    if (epoch == 0 && do_log) {
      debug::log::mainLogger->info("==================================================================================");
      debug::log::mainLogger->info("==================== playing Next match ====================");
      debug::log::mainLogger->info("==================================================================================");
    }
#endif
    if (_goesFist == 0)
    {
      goesFist = -goesFist;
    }

    std::unordered_map<int, Agent*> players = {
      {goesFist, agent1},
      {-goesFist,agent2}
    };
    agent1->getTree()->reset();
    agent2->getTree()->reset();

    auto tmpMemory = memory->getTempMemory();

#if MainLogger
    if (epoch == 0 && do_log)
    {
      debug::log::mainLogger->info("player {} will start",goesFist);
    }
#endif
```

```
    game->reset();
    int turn = 0;
    while (!game->state->done) {
      turn++;
      //std::cout << turn << std::endl;
      auto actionData = players[game->state->player]->getAction(game->state, probMoves > turn);
      tmpMemory.commit(game->state, actionData.second.first);
#if SaverType == 1
      saver.addState(game->state);
#elif SaverType == 2
      saver.addState(actionData.first);
#endif
#if MainLogger
      if (epoch == 0 && do_log) {
        game->state->render(debug::log::mainLogger);
        debug::log::mainLogger->info("MSCT vals: {:1.5f}", actionData.second.second);
        debug::log::logVector(debug::log::mainLogger, actionData.second.first);
        debug::log::mainLogger->info("NN vals: {:1.5f}", players[game->state->player]->predict(game->state).first);
        debug::log::logVector(debug::log::mainLogger, players[game->state->player]->predict(game->state).second);

        debug::log::mainLogger->info("selected action is: {}", actionData.first);
      }
#endif
      game->takeAction(actionData.first);
    }
#if SaverType == 1
    saver.addState(game->state);
#endif
#if MainLogger
    if (epoch == 0 && do_log) {
      game->state->render(debug::log::mainLogger);
    }
#endif
#if ProfileLogger
    debug::Profiler::profiler.switchOperation(4);
#endif
    memory->updateMemory(game->state->player, std::get<0>(game->state->val), &tmpMemory);
    if (true)
```

```
    {
      if (std::get<0>(game->state->val) != 0)
      {
        output->updateValue(players[game->state->player * std::
    get<0>(game->state->val)]);
      }
    }
#if ProfileLogger
    debug::Profiler::profiler.stop();
#endif
  }

#if SaverType == 2 || SaverType == 1
  saver.save(RunId);
#endif

#if RenderGenAndTurneyProgress
  jce::consoleUtils::render_progress_bar(1.0f, true);
#endif
}
```

5.1.12.7.13 utils.hpp

```
#pragma once
#include <ostream>
#include <istream>
#include <vector>

namespace AlphaZero {
  namespace ai {
    // Softams function inp is an iteratable of numbers
    template<typename T>
      // e^i/e^i (softmax function)
    void softmax(T& i);

      // i/i
    template<typename T>

    void linmax(T& i);

    template<typename T>
      //  v
    T getSumm(std::vector<T>& v);
  }
}

template<typename T>
inline void AlphaZero::ai::softmax(T& inp){
  typedef float number;

  number m = -10e100;
  for (number const& z : inp){
    if (m < z) {
      m = z;
    }
  }

  number sum = 0.0;
  for (number const& z : inp) {
    sum += exp(z - m);
  }

  number constant = m + log(sum);
  for (number& z : inp) {
    z = exp(z - constant);
  }
```

```
  throw "Depricated function";
  return;
}

template<typename T>
void AlphaZero::ai::linmax(T& inp)
{
  float sum = 0;
  for (auto const& idx : inp)
  {
    sum = sum + idx;
  }
  for (auto& idx : inp)
  {
    idx = idx / sum;
  }
  return;
}

template<typename T>
T AlphaZero::ai::getSumm(std::vector<T>& val)
{
  T out = 0;
  for (const T& value : val)
  {
    out = out + value;
  }
  return out;
}
```

5.1.12.8 game

5.1.12.8.1 game.hpp

```cpp
#pragma once
/*
The game of connect for coded with game states. This allows The MCTS to simulate the game without changing it.
*/

#include <iostream>
#include <vector>
#include <list>
#include <memory>
#include <tuple>
#include <unordered_map>
#include <bitset>
#include <unordered_set>
#include <torch/torch.h>

#include "config.hpp"

// input_shape is the shape of the game board. x, y is the shape of the actual game board and z is number of stacked planes (in this case one for each
// player).
#define input_shape_x 7
#define input_shape_y 6
#define input_snape_z 2
#define action_count 42
#define action_shape 6, 7
#define boardOfset 42 // the size of a layer of the board in the buffer. (the amount of felds)(x * y)
// the actual name of the game
#define gameName "connect4"


namespace AlphaZero {
  namespace Game {
      // Game State class contains all information of a certain board possition. a board with the positions of all pieces
      // along with the curretn player. It also computes legal Actions A and win information
     class GameState {

```

```cpp
    // the current player 1 or -1
    public: int player;

    // true if game is done (4 in a row or all filled) and false if not.
    public: bool done;

    // winning infromation who won and by how mutch. the tuple contains the following infromation in this order <current player win (1) or
    // current player loose (-0) or tie (0), current player points (1 for win -1 for loose and 0 for tie), other player points (- current       // player points
    public: std::tuple<int, int, int> val;

    // 84 bit bitmap that contains the current board shape. The first 42 bits are the positions of the player ones stones (0 for empy, 1 for
    // stone presant) and the second 42 bits are the same for player -1. The 6x7 board is encoded by placing the 6 rows next to each other
    // starting from the fron so the top left would be bit 0 and the bottom right bit 41.
    public: IDType gameBoard;

    // list of allowed actions. every Action is the index of the bit where the stone would be placed.
    public: std::vector<int> allowedActions;


    // construct from known game state and player.
    public: GameState(IDType board, int _player);

    // construct using default state.
    public: GameState();

    // utilit function called by the constructors to avoid duplicate code. (all the initialization done by both of the constructos)
    private: void initialize(IDType board, int _player);

    // simulate an action from the current game state. (compute the state you would reach from this state by taking the following action)
    public: std::shared_ptr<GameState> takeAction(int action);
```

```cpp

    // check weather the game is done. (will set the done boolean and the val tuple)
    public: void gameIsDone();

    // compute all allowed actions and writes it to the allowed actions list.
    protected: void getAllowedActions();

    // function that will return 1 if player 1 has a stone at the position specified by id, -1 if player 2 does and 0 if nether.
    public: int IdIndex(int id);

    //returns the id of the game state. In this case the game board will surfice as it contains all information. (unable to remove stones)
    public: IDType id();

    // renders the game state to the console in a way that is readable for humans.
    public: void render();
#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger)
    // same as render() but rendes to a specified logger.
    public: void render(std::shared_ptr<spdlog::logger> logger);
#endif
    // sets the piece at id to 0, 1 or -1 (val). b is the board that the piece will be set to. (see gameBoard for encoding)
    public: void static IdIndex(int id, int val, IDType& b);

    // converts the game state to a tensor, that is than passed thought the model.
    public: torch::Tensor toTensor();

    // dose tha same thing but is more efficient for stacked game states during training. It will set tensor[idx] to the tensor representing
    // this game state.
    public: void toTensor(torch::Tensor& tensor, unsigned short idx=0);

    // the character representing the pieca at the position val. (a + if a stone can be placed there)
    private: char getPiece(int val);
```

```
    // recursive function used to determin the height of a colum
     to determin where stones can be placed. The returned int is
    a possible
    // placement position if the bool is true. if the bool is
    false the colum is full.
    private: std::pair<int, bool> getAllowedColumHeight(int);
    };

    // hash function used for hash mapes using the board as a
    hash key.
    struct StateHash
    {
      std::size_t operator()(std::pair<std::shared_ptr<GameState
    >, std::vector<int>> const& s) const noexcept;
    };
    // returns all identidal game states and action maps (N) to
    the passed one.
    std::vector<std::pair<std::shared_ptr<AlphaZero::Game::
    GameState>, std::vector<int>>> identities(std::shared_ptr<
    GameState> state, std::vector<int>& actionProbs);


      // The Actual game contains and handles the gamestates for
     the current generation Game.
    class Game {

    // pointer to the current game state.
    public: std::shared_ptr<GameState> state;


    // constructor will initialize a game state.
    public: Game();

    // reset game to initial position
    public: void reset();

    // take an action as defined by the game state
    public: void takeAction(int action);

    // human action (action is the colum that the stone should
    be placed in, than the action is determined)
    public: bool takeHumanAction(int action);

    // call the states render function.
```

```
    public: void render();
    };

    // Test the game.
    inline void test() {
      AlphaZero::Game::Game* game = new AlphaZero::Game::Game();

      while (!game->state->done) {
        std::vector<int> vec = { 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 };
        auto idents = identities(game->state, vec);
        idents[1].first->render();
        std::cout << "your action: ";
        int action;
        std::cin >> action;
        game->takeHumanAction(action);
#if Windows
        system("cls");
#else
        system("clear");
#endif
      }
      game->render();

      std::cout << std::endl << "the last player just won";
    }
  }
}

inline std::size_t AlphaZero::Game::StateHash::operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>>  const& s) const noexcept {
  return s.first->gameBoard.to_ullong();
}

inline int AlphaZero::Game::GameState::IdIndex(int id)
{
  if (this->gameBoard[id] == 1) {
    return 1;
  }
  else if (this->gameBoard[id + boardOfset] == 1) {
    return -1;
  }
```

```
  return 0;
}

inline void AlphaZero::Game::GameState::IdIndex(int id, int val, IDType& b)
{
  if (val == 0) {
    b.set(id, 0);
    b.set(id + boardOfset, 0);
    return;
  }
  if (val == -1) {
    id += boardOfset;
  }
  b.set(id, 1);
}

inline char AlphaZero::Game::GameState::getPiece(int id)
{
  std::unordered_map<int, char> renderData = { {0,'-'}, {1, 'X'}, {-1, 'O'} };
  if (std::find(this->allowedActions.begin(), this->allowedActions.end(), id) != this->allowedActions.end()) {
    return '+';
  }
  auto va = renderData[this->IdIndex(id)];
  return va;
}

inline IDType AlphaZero::Game::GameState::id()
{
  return this->gameBoard;
}

inline void AlphaZero::Game::Game::render() {
  this->state->render();
}

inline torch::Tensor AlphaZero::Game::GameState::toTensor()
{
  at::Tensor outTensor = at::zeros({ 1, input_snape_z, input_shape_y, input_shape_x });
  this->toTensor(outTensor);
  return outTensor;
}
```

```
inline void AlphaZero::Game::GameState::toTensor(torch::Tensor& tensor, unsigned short idx)
{
  unsigned short pos = 0;
  unsigned int ofset = (this->player == -1) ? 0 : boardOfset;
  for (unsigned short z = 0; z < input_snape_z; z++) {
    for (unsigned short y = 0; y < input_shape_y; y++) {
      for (unsigned short x = 0; x < input_shape_x; x++) {
        tensor[idx][z][y][x] = (float)this->gameBoard[(pos + ofset) % stateSize];
        pos++;
      }
    }
  }
}
```

5.1.12.8.2 game.cpp

```cpp
#include "game.hpp"

#define columOfset 7

AlphaZero::Game::GameState::GameState(IDType board, int _player)
{
  this->initialize(board, _player);
}

AlphaZero::Game::GameState::GameState()
{
  this->initialize(IDType(), 1);
}

void AlphaZero::Game::GameState::initialize(IDType board, int _player)
{
  this->gameBoard = board;
  this->player = _player;
  this->getAllowedActions();
  this->gameIsDone();
}

std::shared_ptr<AlphaZero::Game::GameState> AlphaZero::Game::GameState::takeAction(int action)
{
  IDType newBoard = this->gameBoard;
  GameState::IdIndex(action, this->player, newBoard);

  std::shared_ptr<GameState> newState = std::make_shared<GameState>(newBoard, -this->player);
  return newState;
}

void AlphaZero::Game::GameState::gameIsDone()
{
  std::vector<std::vector<int>> winOptions = {
    /*
    +----+----+----+----+----+----+----+
    | 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 |
    +----+----+----+----+----+----+----+
    */
    //horizontal
    {0,1,2,3},
    {1,2,3,4},
    {2,3,4,5},
    {3,4,5,6},

    {7,8,9,10},
    {8,9,10,11},
    {9,10,11,12},
    {10,11,12,13},

    {14,15,16,17},
    {15,16,17,18},
    {16,17,18,19},
    {17,18,19,20},

    {21,22,23,24},
    {22,23,24,25},
    {23,24,25,26},
    {24,25,26,27},

    {28,29,30,31},
    {29,30,31,32},
    {30,31,32,33},
    {31,32,33,34},

    {35,36,37,38},
    {36,37,38,39},
    {37,38,39,40},
    {38,39,40,41},
    //vertical
    {0 , 7,14,21},
    {7 ,14,21,28},
    {14,21,28,35},

    {1 ,8 ,15,22},
```

```
    {8 ,15,22,29},
    {15,22,29,36},

    {2,9,16,23},
    {9,16,23,30},
    {16,23,30,37},

    {3 ,10,17,24},
    {10,17,24,31},
    {17,24,31,38},

    {4 ,11,18,25},
    {11,18,25,32},
    {18,25,32,39},

    {5 ,12,19,26},
    {12,19,26,33},
    {19,26,33,40},

    {6 ,13,20,27},
    {13,20,27,34},
    {20,27,34,41},

    //diagonal topleft-bottomRight
    {14,22,30,38},

    {7, 15,23,31},
    {15,23,31,39},

    {0, 8, 16,24},
    {8, 16,24,32},
    {16,24,32,40},

    {1, 9 ,17,25},
    {9, 17,25,33},
    {17,25,33,41},

    {2, 10,18,26},
    {10,18,26,34},

    {3, 11,19,27},

    //diagonal topright-bottomleft
    {3, 9, 15,21},

```

```
    {4, 10,16,22},
    {10,16,22,28},

    {5, 11,17,23},
    {11,17,23,29},
    {17,23,29,35},

    {6, 12,18,24},
    {12,18,24,30},
    {18,24,30,36},

    {13,19,25,31},
    {19,25,31,37},

    {20,26,32,38},
  };
  bool tie = true;
  for (int idx = 0; idx < action_count; idx++) {
    if (this->IdIndex(idx) == 0) {
      tie = false;
      break;
    }
  }
  if (tie) {
    this->done = true;
    this->val = { 0,0,0 };
    return;
  }
  for (auto option : winOptions) {
    int count = 0;
    for (int pos : option) {
      count += this->IdIndex(pos);
    }
    if (count == -4 * this->player) {
      this->done = true;
      this->val = { -1, -1, 1 }; // winForThisPlayer, points for this player, points for other player
      return;
    }
  }
  this->done = false;
  this->val = { 0, 0, 0 };
}

inline std::pair<int, bool> AlphaZero::Game::GameState::
```

```
getAllowedColumHeight(int idx) {
  if (this->IdIndex(idx) != 0) {
    return { idx, false };
  }
  if (idx >= 35) {
    return {idx, true};
  }
  else if (this->IdIndex(idx + columOfset)!=0) {
    return { idx, true };
  }
  else {
    return this->getAllowedColumHeight(idx + columOfset);
  }
}

void AlphaZero::Game::GameState::getAllowedActions()
{
  this->allowedActions.clear();
  for (int idx = 0; idx < 7; idx++) {
    std::pair<int, bool> data = this->getAllowedColumHeight(idx);
    if (data.second) {
      this->allowedActions.push_back(data.first);
    }
  }
}

void AlphaZero::Game::GameState::render()
{
  console_mutex.lock();
  for (int row = 0; row < action_count;) {
    for (int iter = 0; iter < 7; iter++) {
      std::cout << this->getPiece(row) << " ";
      row++;
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
  console_mutex.unlock();
}

#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger)
void AlphaZero::Game::GameState::render(std::shared_ptr<spdlog::logger> logger)
```

```
{
  for (int idx = 0; idx < 6; idx++) {
    char line1[13] = {
      this->getPiece(0 + columOfset * idx), ' ',
      this->getPiece(1 + columOfset * idx), ' ',
      this->getPiece(2 + columOfset * idx), ' ',
      this->getPiece(3 + columOfset * idx), ' ',
      this->getPiece(4 + columOfset * idx), ' ',
      this->getPiece(5 + columOfset * idx), ' ',
      this->getPiece(6 + columOfset * idx)
    };
    logger->info(line1);
  }
}
#endif

AlphaZero::Game::Game::Game()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::reset()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::takeAction(int action)
{
  auto newState = this->state->takeAction(action);
  this->state = newState;
}

bool AlphaZero::Game::Game::takeHumanAction(int action)
{
  for (auto const& allowed : this->state->allowedActions) {
    if ((allowed - action) % 7 == 0) {
      this->takeAction(allowed);
      return true;
    }
  }
  return false;
}

inline std::pair<std::shared_ptr<AlphaZero::Game::GameState>,
    std::vector<int>> mirrorGameState(std::shared_ptr<AlphaZero::
```

```
Game::GameState> state, std::vector<int>& actionProbs) {
  IDType boardBuffer;

  std::vector<int> probs = {
     actionProbs[6],  actionProbs[5],  actionProbs[4],  actionProbs[3],  actionProbs[2],  actionProbs[1],  actionProbs[0],
     actionProbs[13], actionProbs[12], actionProbs[11], actionProbs[10], actionProbs[9],  actionProbs[8],  actionProbs[7],
     actionProbs[20], actionProbs[19], actionProbs[18], actionProbs[17], actionProbs[16], actionProbs[15], actionProbs[14],
     actionProbs[27], actionProbs[26], actionProbs[25], actionProbs[24], actionProbs[23], actionProbs[22], actionProbs[21],
     actionProbs[34], actionProbs[33], actionProbs[32], actionProbs[31], actionProbs[30], actionProbs[29], actionProbs[28],
     actionProbs[41], actionProbs[40], actionProbs[39], actionProbs[38], actionProbs[37], actionProbs[36], actionProbs[35]
  };
#define assignStateSinge(idx1, idx2)AlphaZero::Game::GameState::IdIndex(idx1, state->IdIndex(idx2), boardBuffer)
#define assignState(idx1, idx2)assignStateSinge(idx1, idx2); assignStateSinge(idx2, idx1);


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

  return { std::make_shared<AlphaZero::Game::GameState>(boardBuffer, state->player), probs };
```

```
}

std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState
    >, std::vector<int>>> AlphaZero::Game::identities(std::::
    shared_ptr<GameState> state, std::vector<int>& probs)
{
  std::vector<std::pair<std::shared_ptr<AlphaZero::Game::
    GameState>, std::vector<int>>> idents(2);
  int id = 0;
  idents[0] = { state, probs };
  idents[1] = mirrorGameState(state, probs);
  return idents;
}
```

5.1.12.9 jce

5.1.12.9.1 load.hpp

```cpp
#pragma once

#include <fstream>
#include <list>
#include <vector>
#include <bitset>
#include <string>
#include <ai/memory.hpp>
#include <torch/torch.h>
#include <game/game.hpp>

#define BasicLoad(in, data) (in.read((char*)&data, sizeof(data))
    )

namespace jce
{
  // custom part

  // load GameState from file
  void load(std::ifstream& in, std::shared_ptr<AlphaZero::Game::
    GameState>& state);

  // load Memory Element from file
  void load(std::ifstream& in, std::shared_ptr<AlphaZero::ai::
    MemoryElement>& element);

  template<typename key, typename T>
  void load(std::ifstream& in, torch::OrderedDict<key, T>& map);

  // load Tensor from file
  template<typename T=float>
  void load(std::ifstream& in, torch::Tensor& tensor);

  // end custom part

  template<typename key, typename T>
  void load(std::ifstream& in, std::unordered_map<key, T>& map);

  // load pair from file
  template<typename T, typename T2>
  void load(std::ifstream& in, std::pair<T, T2>& data);
```

```cpp

  // load list from file
  template<typename T>
  void load(std::ifstream& in, std::list<T>& data);

  // load vector from file
  template<typename T>
  void load(std::ifstream& in, std::vector<T>& data);

  //load bitset from file
  template<size_t size>
  void load(std::ifstream& in, std::bitset<size>& data);

  // the actuall loading of ints and vectors called by the load
    function
  template <typename T>
  void load_listVec(std::ifstream& in, T& data);

  // load int from file
  void load(std::ifstream& in, int& data);
  // load unsinged int from file
  void load(std::ifstream& in, unsigned int& data);
  // load size_t from file
  void load(std::ifstream& in, size_t& data);
  // load float from file
  void load(std::ifstream& in, float& data);
  // load double from file
  void load(std::ifstream& in, double& data);
  // load char from file
  void load(std::ifstream& in, char*& data);
  // load int64_t from file
  void load(std::ifstream& in, int64_t& data);
  // load string from file
  void load(std::ifstream& in, std::string& data);
}

inline void jce::load(std::ifstream& in, std::shared_ptr<
    AlphaZero::Game::GameState>& state)
{
  IDType board;
  int player;
  jce::load(in, board);
  jce::load(in, player);
  state.reset(new AlphaZero::Game::GameState(board, player));
}
```

```cpp

inline void jce::load(std::ifstream& in, std::shared_ptr<
    AlphaZero::ai::MemoryElement>& element)
{
  element.reset(new AlphaZero::ai::MemoryElement());
  jce::load(in, element->value);
  jce::load(in, element->state);
  jce::load(in, element->av);
}

template<typename key, typename T>
inline void jce::load(std::ifstream& in, torch::OrderedDict<key,
    T>& map)
{
  size_t size;
  jce::load(in, size);
  for (size_t idx = 0; idx < size; idx++)
  {
    key _key;
    T _item;
    jce::load(in, _key);
    jce::load(in, _item);
    map.insert(_key, _item);
  }
}

template<typename T>
inline void jce::load(std::ifstream& in, torch::Tensor& tensor)
{
  std::vector<int64_t> vec;
  int64_t fullSize = 1;
  jce::load(in, vec);
  for (size_t value : vec) { fullSize *= value; }
  tensor = torch::zeros({ fullSize });
  T val, last;
  for (size_t idx = 0; idx < fullSize; idx++)
  {
    jce::load(in, val);
    tensor[idx] = val;
    last = val;
  }
  tensor = torch::reshape(tensor, vec);
}

template<typename key, typename T>
```

```cpp
inline void jce::load(std::ifstream& in, std::unordered_map<key,
    T>& map)
{
  size_t size;
  jce::load(in, size);
  for (size_t idx = 0; idx < size; idx++)
  {
    key _key;
    T _item;
    jce::load(in, _key);
    jce::load(in, _item);
    map.insert({ _key, _item });
  }
}

template<typename T, typename T2>
inline void jce::load(std::ifstream& in, std::pair<T, T2>& data)
{
  jce::load(in, data.first);
  jce::load(in, data.second);
}

template<typename T>
inline void jce::load(std::ifstream& in, std::list<T>& data)
{
  load_listVec(in, data);
}

template<typename T>
inline void jce::load(std::ifstream& in, std::vector<T>& data)
{
  load_listVec(in, data);
}

template<size_t size>
inline void jce::load(std::ifstream& in, std::bitset<size>& data
    )
{
  char byte;
  for (size_t idx = 0; idx < size; idx = idx + 8)
  {
    in.read(&byte, 1);
    std::bitset<8> tempSet(byte);
    for (size_t pos = 0; pos < 8 && pos + idx < size; pos++)
    {
```

```cpp
      data.set(pos + idx, tempSet[pos]);
    }
  }
}

template<typename T>
inline void jce::load_listVec(std::ifstream& in, T& data)
{
  size_t size;
  jce::load(in, size);
  data.resize(size);
  for (auto& val : data)
  {
    jce::load(in, val);
  }
}

inline void jce::load(std::ifstream& in, int& data){ BasicLoad(
    in, data); }
inline void jce::load(std::ifstream& in, unsigned int& data) {
    BasicLoad(in, data); }
inline void jce::load(std::ifstream& in, size_t& data) {
    BasicLoad(in, data); }
inline void jce::load(std::ifstream& in, float& data) {
    BasicLoad(in, data); }
inline void jce::load(std::ifstream& in, double& data) {
    BasicLoad(in, data); }
inline void jce::load(std::ifstream& in, int64_t& data) {
    BasicLoad(in, data); }

inline void jce::load(std::ifstream& in, std::string& data)
{
  char* c_arr;
  jce::load(in, c_arr);
  data = std::string(c_arr);
}

inline void jce::load(std::ifstream& in, char*& data)
{
  std::vector<char> data_vec;
  while (true)
  {
    char next;
    in.read(&next, 1);
    data_vec.push_back(next);
```

```
    if (next == NULL)
    {
      break;
    }
  }
  data = new char[data_vec.size()];
  auto pos = data;
  for (auto const& value : data_vec)
  {
    (*pos) = value;
    pos++;
  }
}
```

5.1.12.9.2 save.hpp

```
#pragma once

#include <fstream>
#include <iostream>
#include <list>
#include <vector>
#include <string>
#include <bitset>
#include <ai/memory.hpp>
#include <torch/torch.h>

#define BasicSave(data, out) (out.write((char*)&data, sizeof(data)))
#define BasicSave_cp(data, out) (out.write((char*)data, sizeof(data)))

namespace jce
{
  // custom part

  // save GameState to file
  void save(std::ofstream& out, std::shared_ptr<AlphaZero::Game::GameState> const& state);

  // save memory element to file
  void save(std::ofstream& out, std::shared_ptr<AlphaZero::ai::MemoryElement> const& element);

  // save Tensor to file
  template<typename T=float>
  void save(std::ofstream& out, torch::Tensor const& tensor);

  template<typename key, typename T>
  void save(std::ofstream& out, torch::OrderedDict<key, T> const& map);

  // end custom part

  template<typename T, typename key>
  void save(std::ofstream& out, std::unordered_map<T, key>);

  template<typename T, typename T2>
  // save pair to file
```

```
  void save(std::ofstream& out, std::pair<T, T2>const& data);

  template<typename T>
  //save list to ofstream
  void save(std::ofstream& out, std::list<T> const& data);

  template<typename T>
  //save vector to file
  void save(std::ofstream& out, std::vector<T> const& data);
  template<typename T>
  void quick_save(std::ofstream& out, std::vector<T> const& data
    );

  //save bitset to file
  template <size_t T>
  void save(std::ofstream& out, std::bitset<T> const& data);

  template<typename T>
  //the actuall saving function for vectors and lists
  void save_listVec(std::ofstream& out, T const& data);

  // save int to file
  void save(std::ofstream& out, int const& data);
  // save unsigned int to file
  void save(std::ofstream& out, unsigned int const& data);
  // save size_t to to file
  void save(std::ofstream& out, size_t const& data);
  // save float to file
  void save(std::ofstream& out, float const& data);
  // save double to file
  void save(std::ofstream& out, double const& data);
  // save char to file
  void save(std::ofstream& out, const char* arr);
  // save int64_t to file
  void save(std::ofstream& out, const int64_t& data);
  // save string to file
  void save(std::ofstream& out, const std::string& data);
}

inline void jce::save(std::ofstream& out, std::shared_ptr<
    AlphaZero::ai::MemoryElement> const& element)
{
  jce::save(out, element->value);
  jce::save(out, element->state);
  jce::save(out, element->av);
```

```cpp
}

template<typename T>
inline void jce::save(std::ofstream& out, torch::Tensor const&
    tensor)
{
  jce::save(out, tensor.sizes().vec());
  auto flatTensor = torch::flatten(tensor);
  for (size_t idx = 0; idx < flatTensor.size(0); idx++)
  {
    T val = flatTensor[idx].item<T>();
    jce::save(out, val);
  }
}

template<typename key, typename T>
inline void jce::save(std::ofstream& out, torch::OrderedDict<key
    , T> const& map)
{
  jce::save(out, map.size());
  for (auto& val : map)
  {
    jce::save(out, val.key());
    jce::save(out, val.value());
  }
}

inline void jce::save(std::ofstream& out, std::shared_ptr<
    AlphaZero::Game::GameState> const& state)
{
  jce::save(out, state->gameBoard);
  jce::save(out, state->player);
}

template<typename T, typename key>
inline void jce::save(std::ofstream& out, std::unordered_map<T,
    key> map)
{
  jce::save(out, map.size());
  for (auto const& val : map)
  {
    jce::save(out, val.first);
    jce::save(out, val.second);
  }
}
```

```cpp

template<typename T, typename T2>
inline void jce::save(std::ofstream& out, std::pair<T, T2> const
    & data)
{
  jce::save(out, data.first);
  jce::save(out, data.second);
}

template<typename T>
inline void jce::save(std::ofstream& out, std::list<T> const&
    data)
{
  save_listVec(out, data);
}

template <typename T>
inline void jce::quick_save(std::ofstream& out, std::vector<T>
    const& data)
{
  jce::save(out, data.size());
  BasicSave_cp(data.data(), out);
}

template <typename T>
inline void jce::save(std::ofstream& out, std::vector<T> const&
    data)
{
  save_listVec(out, data);
}

template<size_t T>
inline void jce::save(std::ofstream& out, std::bitset<T> const&
    data)
{
  std::bitset<8> temp;
  size_t tempVal;
  for (size_t idx = 0; idx < T; idx = idx + 8)
  {
    for (size_t pos = 0; pos < 8 && pos + idx < T; pos++) {
      temp.set(pos, data[pos + idx]);
    }
    tempVal = temp.to_ullong();
    out.write((char*)&tempVal, 1);
  }
```

```
}

template <typename T>
inline void jce::save_listVec(std::ofstream& out, T const & data)
{
  jce::save(out, data.size());
  for (auto const& data : data)
  {
    jce::save(out, data);
  }
}

inline void jce::save(std::ofstream& out, int const& data) { BasicSave(data, out); }
inline void jce::save(std::ofstream& out, unsigned int const& data) { BasicSave(data, out); }
inline void jce::save(std::ofstream& out, size_t const& data) { BasicSave(data, out); }
inline void jce::save(std::ofstream& out, float const& data) { BasicSave(data, out); }
inline void jce::save(std::ofstream& out, double const& data) { BasicSave(data, out); }
inline void jce::save(std::ofstream& out, const int64_t& data) { BasicSave(data, out); }

inline void jce::save(std::ofstream& out, const std::string& data)
{
  jce::save(out, data.c_str());
}

inline void jce::save(std::ofstream& out, const char* arr)
{
  size_t size = 1;
  auto iterator = arr;
  while (*iterator != NULL)
  {
    size++;
    iterator++;
  }
  out.write(arr, size);
}
```

5.1.12.9.3 string.hpp

```cpp
#pragma once
#include <iostream>

namespace jce
{
    namespace consoleUtils
    {
            // render a progress bar to console (from stack overflow)
        void render_progress_bar(float progress, bool persistand = false);
    }
}

inline void jce::consoleUtils::render_progress_bar(float progress, bool persistant)
{
#if true
    if (progress <= 1.0) {
        int barWidth = 70;

        std::cout << "[";
        int pos = barWidth * progress;
        for (int i = 0; i < barWidth; ++i) {
            if (i < pos) std::cout << "=";
            else if (i == pos) std::cout << ">";
            else std::cout << " ";
        }
        if (persistant)
        {
            std::cout << "] " << int(progress * 100.0) << std::endl;
        }
        else
        {
            std::cout << "] " << int(progress * 100.0) << " %\r";
        }
        std::cout.flush();
    }
#endif
}
```

5.1.12.9.4 vector.hpp

```
#pragma once
#include <vector>

namespace jce {
  namespace vector {
    template <typename T>
      // eqivilent to np.fill() [one dimension only]
    std::vector<T> gen(size_t size, T val);
  }
}

template<typename T>
inline std::vector<T> jce::vector::gen(size_t size, T val)
{
  std::vector<T> out(size);
  for (auto& item : out) {
    item = val;
  }
  return out;
}
```

5.1.12.10 server

5.1.12.10.1 eloClient.hpp

```
// client that interfaces with the elo server (elo directory)

#pragma once

#include <config.hpp>
#include <sockpp/tcp_connector.h>
#include <ai/agent.hpp>


#define ELO_PORT 2551
#define ELO_IP "wandhoven.ddns.net"

namespace AlphaZero
{
  namespace elo
  {
      // Class that handles communication to the elo server.
    class eloClient
    {
      // set the elo rating of a certain agent
    public: int setElo(int agent1, int eloRating) const;

      // get the elo rating of an agent.
    public: int getElo(int agent1) const;

      // get the agent with the clossest elo rating to the
     specified value.
    public: int getAgentWithClosestElo(int eloVal) const;
    };
  }
}

inline int AlphaZero::elo::eloClient::setElo(int agent1, int elo
    ) const
{
  sockpp::socket_initializer sockInit;
  in_port_t port = ELO_PORT;
  std::string host = ELO_IP;
  sockpp::tcp_connector conn({ host, port });
  if (!conn)
  {
```

```
    std::cout << (conn.last_error_str()) << std::endl;
    return -1;
  }

  int data[3] = { 2, agent1, elo };
  conn.write(data, sizeof(data));

  int delo[1];
  conn.read(delo, sizeof(delo));
  return delo[0];
}

inline int AlphaZero::elo::eloClient::getElo(int agent1) const
{
  sockpp::socket_initializer sockInit;
  in_port_t port = ELO_PORT;
  std::string host = ELO_IP;
  sockpp::tcp_connector conn({ host, port });
  if (!conn)
  {
    std::cout << (conn.last_error_str()) << std::endl;
    return -1;
  }

  int data[2] = { 1, agent1 };
  conn.write(data, sizeof(data));

  int elo[1];
  conn.read(elo, sizeof(elo));
  return elo[0];
}

inline int AlphaZero::elo::eloClient::getAgentWithClosestElo(int
    val) const
{
  sockpp::socket_initializer sockInit;
  in_port_t port = ELO_PORT;
  std::string host = ELO_IP;
  sockpp::tcp_connector conn({ host, port });
  if (!conn)
  {
    std::cout << (conn.last_error_str()) << std::endl;
    return -1;
  }

```

```
  int data[2] = { -1, val };
  conn.write(data, sizeof(data));

  int elo[1];
  conn.read(elo, sizeof(elo));
  return elo[0];
}
```

5.1.12.10.2 server.hpp

```cpp
// Ai server

#pragma once

#include <config.hpp>
#include <sockpp/tcp_acceptor.h>
#include <ai/agent.hpp>


#define PORT 25500

namespace AlphaZero
{
  namespace Server
  {
      // convert an array socket communication to board
    IDType toBoard(int arr[]);

      // Ai communication server
    class TCPServer
    {
      // evaluate a socket request (ask the Ai what action they want to take and send it back)
    private: void evaluate(sockpp::tcp_socket& sock);

      // socket initializer (see https://github.com/fpagliughi/sockpp)
    private: sockpp::socket_initializer sockInit;

      // server listening socket used to accept evaluation requests
    private: sockpp::tcp_acceptor acc;

      // accept an incomming connection, call the evaluate method and close the connection
    private: void accept();


      // server constructor on a certain port
    public: TCPServer(int port = PORT);

      // while true call accept()
    public: void mainLoop();
```

```
    };

      // same as the TCPServer but uses human inputs for tesing purpusses.
    class TestServer
    {
      // socket initializer (see https://github.com/fpagliughi/sockpp)
    private: sockpp::socket_initializer sockInit;

      // server listening socket used to accept evaluation requests
    private: sockpp::tcp_acceptor acc;

      // accet connection, ask human controller for action, send action to client, close connection
    private: void accept();


      // create a test server on a certain port
    public: TestServer(int port = PORT);

      // call the accept method while forever
    public: void mainLoop();
    };
  }
}

inline IDType AlphaZero::Server::toBoard(int arr[])
{
  IDType out;
  for (int idx = 0; idx < stateSize; idx++)
  {
    out.set(idx, arr[idx]);
  }
  return out;
}
```

5.1.12.10.3 server.cpp

```
// SockServer.cpp : Defines the entry point for the application.
//

#pragma comment( lib, "ws2_32.lib")

#include "server.hpp"
#include <game/game.hpp>
#include <iostream>
#include <log.hpp>


std::shared_ptr<spdlog::logger> logger = debug::log::
    createLogger("ServerLogger", "logs/c++/Server.log");

std::vector<char*> devices = { DEVICES };
std::shared_ptr<AlphaZero::ai::Agent> agent = std::make_shared<
    AlphaZero::ai::Agent>(devices);

inline void AlphaZero::Server::TCPServer::evaluate(sockpp::
    tcp_socket& sock) {
  ssize_t n;
  int buf[stateSize + 2];
  int out[1];

  n = sock.read(buf, sizeof(buf));

  std::shared_ptr<AlphaZero::Game::GameState> state = std::
    make_shared<AlphaZero::Game::GameState>(AlphaZero::Server::
    toBoard(buf), buf[stateSize]);

  agent->reset();
  try
  {
    std::cout << "model version is: " << buf[stateSize + 1] <<
    std::endl;
    agent->model->load_version(buf[stateSize + 1]);
  }
  catch (...)
  {
    agent->model->load_current();
  }

  auto actionData = agent->getAction(state, false);
```

```
  out[0] = actionData.first;

#if MainLogger
  state->render(logger);
  logger->info("MSCT vals: {:1.5f}", actionData.second.second);
  debug::log::logVector(logger, actionData.second.first);
  logger->info("NN vals: {:1.5f}", agent->predict(state).first);
  debug::log::logVector(logger, agent->predict(state).second);
  logger->info("NN Q:");
  debug::log::logVector(logger, AlphaZero::ai::getQ(agent->getTree()->getNode(state->id())));

  logger->info("selected action is: {}", actionData.first);

  logger->flush();
#endif

  sock.write_n(out, sizeof(int));

  logger->info("Connection closed");
}

void AlphaZero::Server::TCPServer::accept()
{
  sockpp::tcp_socket sock = this->acc.accept();
  logger->info("Connection acceptd from ", sock.peer_address().to_string());
  evaluate(sock);
}


AlphaZero::Server::TCPServer::TCPServer(int _port)
{
  in_port_t port = _port;
  this->acc = sockpp::tcp_acceptor(port);

  if (!acc) {
    std::cerr << "Error creating the acceptor: " << acc.last_error_str() << std::endl;
  }

  std::cout << "Acceptor bound to address: " << this->acc.address() << std::endl;
  std::cout << "Awaiting connections on port " << port << "..." << std::endl;
```

```

  logger->info("Acceptor bound to address: ", this->acc.address
    ().to_string());
  logger->info("Awaiting connections on port: {}", port);
}

void AlphaZero::Server::TCPServer::mainLoop()
{
  while (true)
  {
    this->accept();
  }
}

AlphaZero::Server::TestServer::TestServer(int _port)
{
  in_port_t port = _port;
  this->acc = sockpp::tcp_acceptor(port);

  if (!acc) {
    std::cerr << "Error creating the acceptor: " << acc.
    last_error_str() << std::endl;
  }
  std::cout << "Acceptor bound to address: " << acc.address() <<
     std::endl;
  std::cout << "Awaiting connections on port " << port << "..."
   << std::endl;
}

void AlphaZero::Server::TestServer::mainLoop()
{
  while (true)
  {
    this->accept();
  }
}

void AlphaZero::Server::TestServer::accept()
{
  sockpp::tcp_socket sock = this->acc.accept();

  std::cout << "Connection acceptd from " << sock.peer_address()
    << std::endl;

  ssize_t n;
```

```cpp
  int buf[stateSize + 1];
  int out[1];

  n = sock.read(buf, sizeof(buf));

  std::shared_ptr<Game::GameState> state = std::make_shared<Game
    ::GameState>(toBoard(buf), buf[stateSize]);
  state->render();

  std::cout << "Server Action for testing: ";
  std::cin >> out[0];
  std::cout << std::endl;

  sock.write_n(out, sizeof(int));

  std::cout << "Connection closed from " << sock.peer_address()
   << std::endl;
}
```

5.1.12.11 test

5.1.12.11.1 testSuit.hpp

```cpp
// file used for testing that everything works (mostly debugging)

#include <ai/model.hpp>
#include <ai/agent.hpp>
#include "testUtils.hpp"

namespace AlphaZero
{
  namespace test
  {
      // call all the individual tests
    void runTests();

      // test model coppying
    void testCoppying();

      // test standard model saving (pytorch saving)
    void testSave();

      // test backup saving method (jce)
    void testJCESave();

      // test the loss logger to ensure it works
    void testLossLog();

      // test that the model prediction works with multiple states at the same time
    void testModelData();

      // test if training works
    void testTraining();

      // test the prediction speed of the model
    void testModelSpeed();

      // test that the model synchonizes (depricated)
    void testModelSyncronization();

      // check that the prediction of two agents dose the same thing.
```

```cpp
    bool compareAgents(std::shared_ptr<ai::Agent> anget1, std::shared_ptr<ai::Agent> anget2);
    std::shared_ptr<Game::GameState> getRandomState();
  }
}

inline bool AlphaZero::test::compareAgents(std::shared_ptr<ai::Agent> anget1, std::shared_ptr<ai::Agent> anget2)
{
  auto state = getRandomState();

  auto valsA = anget1->predict(state);
  auto valsB = anget2->predict(state);

  if (valsA.first != valsB.first) { return false; }
  for (size_t idx = 0; idx < action_count; idx++)
  {
    if (valsA.second[idx] != valsB.second[idx]) { return false; }
  }
  return true;
}


inline std::shared_ptr<AlphaZero::Game::GameState> AlphaZero::test::getRandomState()
{
  std::bitset<stateSize> board;
  for (size_t idx = 0; idx < stateSize; idx++)
  {
    board.set(idx, rand() % 2);
  }
  auto state = std::make_shared<Game::GameState>(board, rand() % 2);
  return state;
}
```

5.1.12.11.2 testSuit.cpp

```
#include "testSuit.hpp"
#include <stdio.h>
#include <ai/memory.hpp>
#include <ai/modelSynchronizer.hpp>
#include <ai/playGame.hpp>
#include <timer.hpp>

std::vector<char*> devices = { DEVICES };

void AlphaZero::test::runTests()
{
  std::cout << "running Test" << std::endl;
  testModelData();
  testCoppying();
  testSave();
  testJCESave();
  testLossLog();
  testModelSyncronization();
  if (torch::cuda::cudnn_is_available() || randomModel)
    testModelSpeed();
}


void AlphaZero::test::testCoppying()
{
  std::cout << "Testing Model coppying ...\t\t";

  auto modelA = std::make_shared<ai::Agent>(devices);
  auto modelB = std::make_shared<ai::Agent>(devices);

  modelA->model->save_as_current();

  modelB->model->copyModel(modelA->model.get());
  printSuccess(compareAgents(modelA, modelB));
}

void AlphaZero::test::testSave()
{
  std::cout << "Testing Model save ...\t\t\t";

  auto modelA = std::make_shared<ai::Agent>(devices);
  auto modelB = std::make_shared<ai::Agent>(devices);

```

```
  char folder[] = "temp.torch";
  modelA->model->save_to_file(folder);
  modelB->model->load_from_file(folder);

  remove("temp.torch");

  printSuccess(compareAgents(modelA, modelB));
}

void AlphaZero::test::testJCESave()
{
  std::cout << "Testing Model jce save ...\t\t";

  auto modelA = std::make_shared<ai::Agent>(devices);
  auto modelB = std::make_shared<ai::Agent>(devices);

  char folder[] = "temp.torch";
  modelA->model->jce_save_current(folder);
  modelB->model->jce_load_from_file(folder);

  remove("temp.torch");

  printSuccess(compareAgents(modelA, modelB));
}

void AlphaZero::test::testLossLog()
{
  std::cout << "Testing loss Logger ...\t\t\t";
#if LossLogger
  auto log1 = debug::log::lossLogger();
  log1.addValue(1.0f, 2.3f);
  log1.addValue(5.234f, 9834.2345789f);
  log1.newBatch();
  log1.addValue({ 44.634f, 234.4344f });

  char folder[] = "temp.log.bin";
  log1.save(folder);
  auto log2 = debug::log::lossLogger(folder);
  remove(folder);

  printSuccess(log2 == log1);
#else
  std::cout << "\33[1;33mDeactivated\33[0m" << std::endl;
#endif
}
```

```cpp

void AlphaZero::test::testModelData()
{
  std::cout << "Testing model prediction ...\t\t";
  float epsilon = 0.000001f;
  auto model = std::make_shared<ai::Agent>(devices);
  auto states = std::vector<std::shared_ptr<Game::GameState>>({
    getRandomState(), getRandomState(),getRandomState(),
    getRandomState(),getRandomState(),getRandomState(),
    getRandomState(),getRandomState(),getRandomState(),
    getRandomState() });

  ai::ModelSynchronizer syncher(devices);

  auto nodes = std::vector<ai::Node*>();
  auto data = std::list<ai::ModelData*>();
  auto holders = std::vector<std::thread>();

  for (auto const& state : states)
  {
    auto node = new ai::Node(state);
    nodes.push_back(node);
    data.push_back(new ai::ModelData(node));
    holders.push_back(std::thread(ModelSynchronizer::
    _addTestData, data.back(), &syncher));
  }

  auto iter = data.begin();
  bool isValid = true;
  for (auto& holder : holders)
  {
    holder.join();
  }
  for (size_t idx = 0; idx < data.size(); idx++)
  {
    auto a = model->predict(states[idx]);
    auto error = torch::mse_loss(torch::from_blob(a.second.data
    (), a.second.size()), (*iter)->polys);
    //std::cout << error << std::endl;
    iter++;
  }

  printSuccess(isValid);
}

```

```
void AlphaZero::test::testTraining()
{
  auto model = std::make_shared<ai::Agent>(devices);
  auto state = getRandomState();
  auto vec = jce::vector::gen(42, 0);
  vec[0] = 1;
  std::cout << model->model->predict(state) << std::endl;
  std::shared_ptr<ai::Memory> memory = std::make_shared<ai::Memory>();
  for (size_t loop = 0; loop < 10; loop++)
  {
    ai::TemporaryMemory tmpMem(true);
    while (tmpMem.tempMemory.size() < Training_batch * Training_loops)
    {
      //state = getRandomState();
      tmpMem.commit(state, vec);
    }
    memory->updateMemory(0, 0, &tmpMem);
    model->fit(memory, Training_loops);
  }
  std::cout << model->model->predict(state) << std::endl;
  return;
}

void AlphaZero::test::testModelSpeed()
{
  std::cout << "testing Prediction speed ...\t\t";
  std::shared_ptr<ai::Memory> memory = std::make_shared<ai::Memory>();
  std::shared_ptr<ai::Agent> bestAgent = std::make_shared<ai::Agent>(devices);
  std::shared_ptr<Game::Game> game = std::make_shared<Game::Game>();
  char nameBuff[100];
  utils::Timer timer;
  timer.reset();
  auto score = ai::playGames_inThreads(game.get(), bestAgent.get(), bestAgent.get(), memory.get(), Turnement_probabiliticMoves, TurneyEpochs, TurneyThreads, nameBuff, 0, true);
  std::cout << timer.elapsed() << std::endl;
}

void AlphaZero::test::testModelSyncronization()
```

```cpp
{
  std::cout << "testing Model Synchronization ...\t";
  std::vector<char*> devices = { "cpu", "cpu"};
  std::shared_ptr<ai::Agent> bestAgent = std::make_shared<ai::Agent>(devices);

  auto state = getRandomState();

  auto valsA = bestAgent->model->predict(state, 0);
  auto valsB = bestAgent->model->predict(state, 1);

  bool isValid = true;
  if (valsA.first != valsB.first) { isValid = false; }
  if (!torch::equal(valsA.second, valsB.second)) { isValid = false; }
  printSuccess(isValid);
}
```

5.1.12.11.3 testUtils.hpp

```
#pragma once
#include <iostream>

namespace AlphaZero
{
  namespace test
  {
      // print the success or fail message in color
    void printSuccess(bool val);
  }
}

inline void AlphaZero::test::printSuccess(bool val)
{
  if (val)
  {
    std::cout << "\33[32;1mSuccess\33[0m" << std::endl;
  }
  else
  {
    std::cout << "\33[31;1mFailed\33[0m" << std::endl;
  }
}
```

5.2 Clients

5.2.1 ConsoleClient

5.2.1.1 ConsoleClient.h

```
// auto generated file (forgot to remove it)

#pragma once

#include <iostream>
```

5.2.1.2 ConsoleClient.cpp

```cpp
// Console client for alpha Zero (first test client)
//

#include <agent.hpp>

using namespace std;

// play game and get actions form the agents. effectively call
    the game clients and ask the user for an input.
void playGame(std::shared_ptr<Agents::Agent>agent1, std::
    shared_ptr<Agents::Agent>agent2, std::shared_ptr<AlphaZero::
    Game::Game>game)
{
  while (!game->state->done)
  {
    int action;
    switch (game->state->player)
    {
    case(1):{
      action = agent1->getAction(game);
      break;
    }
    case(-1): {
      action = agent2->getAction(game);
      break;
    }
    }
    game->takeAction(action);
  }
}

// create a game, create a user, connect to the AI server and
    start playing games
int main()
{
  auto game = std::make_shared<AlphaZero::Game::Game>();
  auto user = std::make_shared<Agents::User>();
  auto AI = std::make_shared<Agents::RemoteAgent>("35.240.231.50
    ", 12345);
  playGame(user, AI, game);
  return 0;
}
```

5.2.1.3 include

5.2.1.3.1 agent.hpp

```
#pragma once
#pragma comment( lib , "ws2_32.lib")

#include "game.hpp"
#include "modifications.hpp"
#include <string>
#include <sockpp/tcp_connector.h>

namespace Agents
{
  // superclass used to implement user and RemoteAgent
  class Agent
  {
      // virtual method used to define how a subagent computes
    an action
  public: virtual int getAction(std::shared_ptr<AlphaZero::Game
    ::Game> game) = 0;
  };

    // human (console) version of the agent, used to render the
    game to the console and read the input from the console.
  class User : public Agent
  {

    // implmentation of the Agents getAction method used to
    render the game to console and read the action.
  public: virtual int getAction(std::shared_ptr<AlphaZero::Game
    ::Game> game);

    // recursive function used to determine the action and
    ensure its legality.
  private: int subGetAction(std::shared_ptr<AlphaZero::Game::
    Game> game);
  };

  class RemoteAgent : public Agent
  {
    // socket initializer (see sockpp: https://github.com/
    fpagliughi/sockpp)
  private: sockpp::socket_initializer sockInit;

```

```
    // the server ip (domain name) of the server to get the action from
  private: std::string ip;

    // the port the server is listening on.
  private: in_port_t port;

    // set the ip and port of the server and create the initializer.
  public: RemoteAgent(std::string host, in_port_t port);

    // implementation of the Agents getAction mehtod used to request the action from the server.
  public: virtual int getAction(std::shared_ptr<AlphaZero::Game::Game> game);

    // convert the board to an intager array (the array is 1 where the gameboard is 1 and 0 where the gameboard is 0, the last position is equivilent to the player)
  public: void toArr(int* arr, std::shared_ptr<AlphaZero::Game::Game> game);
  };
}

inline int Agents::User::getAction(std::shared_ptr<AlphaZero::Game::Game> game)
{
#if WIN32
  system("cls");
#else
  system("clear");
#endif
  game->render();
  modifications::bottomLable();
  return this->subGetAction(game);
}

// get the character representation of the player (X for 1 and O for -1)
inline char currentPlayerIcon(int player)
{
  switch(player)
  {
  case(1): {return 'X'; };
  case(-1): {return 'O'; };
```

```cpp
  default: {return 'E'; }
  }
}

inline int Agents::User::subGetAction(std::shared_ptr<AlphaZero::Game::Game> game)
{
  std::cout << std::endl << "Move for " << currentPlayerIcon(game->state->player) << ": ";
  int res;
  try {
    std::cin >> res;
  }
  catch (...){
    return this->subGetAction(game);
  }
  for (auto const& val : game->state->allowedActions)
  {
    if (res == modifications::allowedActionModification(val))
    {
      return val;
    }
  }
  std::cin.clear();
  std::cin.ignore(INT_MAX);
  std::cout << std::endl << "\33[31;1mIllegal Move try again\33[0m" << std::endl;
  return this->subGetAction(game);
}

inline Agents::RemoteAgent::RemoteAgent(std::string _host, in_port_t port)
{
  this->ip = _host;
  this->port = port;
  this->sockInit = sockpp::socket_initializer();
}

inline int Agents::RemoteAgent::getAction(std::shared_ptr<AlphaZero::Game::Game> game)
{
  int arr[GameBoardHolderSize + 1];
  this->toArr(arr, game);

  sockpp::tcp_connector con({ this->ip, this->port });
```

```cpp
  con.write(arr, (GameBoardHolderSize + 1) * sizeof(int));

  int out[1];
  con.read_n(out, sizeof(int));
  return out[0];
}

inline void Agents::RemoteAgent::toArr(int* arr, std::shared_ptr<AlphaZero::Game::Game> game)
{
  for (int idx = 0; idx < GameBoardHolderSize; idx++)
  {
    arr[idx] = game->state->gameBoard.test(idx);
  }
  arr[GameBoardHolderSize] = game->state->player;
}
```

5.2.1.3.2 game.hpp

```cpp
#pragma once
/*
The game of connect for coded with game states. This allows The MCTS to simulate the game without changing it.
*/

#include <iostream>
#include <vector>
#include <list>
#include <memory>
#include <tuple>
#include <unordered_map>
#include <bitset>
#include <unordered_set>

// input_shape is the shape of the game board. x, y is the shape of the actual game board and z is number of stacked planes (in this case one for each
// player).
#define input_shape_x 7
#define input_shape_y 6
#define input_snape_z 2
#define action_count 42
#define action_shape 6, 7
#define boardOfset 42 // the size of a layer of the board in the buffer. (the amount of felds)(x * y)
// the actual name of the game
#define gameName "connect4"

#define GameBoardHolderSize 84

namespace AlphaZero {
  namespace Game {
    class GameState {
    // Game State class contains all information of a certain board possition. a board with the positions of all pieces
    // along with the curretn player. It also computes legal Actions A and win information

    // the current player 1 or -1
    public: int player;

    // true if game is done (4 in a row or all filled) and false
```

```
 if not.
    public: bool done;

    // winning infromation who won and by how mutch. the tuple contains the following infromation in this order <current player win (1) or
    // current player loose (-0) or tie (0), current player points (1 for win -1 for loose and 0 for tie), other player points (- current player points)
    public: std::tuple<int, int, int> val;

    // 84 bit bitmap that contains the current board shape. The first 42 bits are the positions of the player ones stones (0 for empy, 1 for
    // stone presant) and the second 42 bits are the same for player -1. The 6x7 board is encoded by placing the 6 rows next to each other
    // starting from the fron so the top left would be bit 0 and the bottom right bit 41.
    public: IDType gameBoard;

    // list of allowed actions. every Action is the index of the bit where the stone would be placed.
    public: std::vector<int> allowedActions;


    // construct from known game state and player.
    public: GameState(IDType board, int _player);

    // construct using default state.
    public: GameState();

    // utilit function called by the constructors to avoid duplicate code. (all the initialization done by both of the constructos)
    private: void initialize(IDType board, int _player);

    // simulate an action from the current game state. (compute the state you would reach from this state by taking the following action)
    public: std::shared_ptr<GameState> takeAction(int action);

    // check weather the game is done. (will set the done boolean and the val tuple)
    public: void gameIsDone();
```

```cpp

    // compute all allowed actions and writes it to the allowed actions list.
    protected: void getAllowedActions();

    // function that will return 1 if player 1 has a stone at the position specified by id, -1 if player 2 does and 0 if nether.
    public: int IdIndex(int id);

    //returns the id of the game state. In this case the game board will surfice as it contains all information. (unable to remove stones)
    public: IDType id();

    // renders the game state to the console in a way that is readable for humans.
    public: void render();
#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger)
    // same as render() but rendes to a specified logger.
    public: void render(std::shared_ptr<spdlog::logger> logger);
#endif
    // sets the piece at id to 0, 1 or -1 (val). b is the board that the piece will be set to. (see gameBoard for encoding)
    public: void static IdIndex(int id, int val, IDType& b);

    // the character representing the pieca at the position val. (a + if a stone can be placed there)
    private: char getPiece(int val);

    // recursive function used to determin the height of a colum to determin where stones can be placed. The returned int is a possible
    // placement position if the bool is true. if the bool is false the colum is full.
    private: std::pair<int, bool> getAllowedColumHeight(int);
    };

    // hash function used for hash mapes using the board as a hash key.
    struct StateHash
    {
      std::size_t operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>> const& s) const noexcept;
```

```
};
// returns all identidal game states and action maps (N) to the passed one.
std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState>, std::vector<int>>> identities(std::shared_ptr<GameState> state, std::vector<int>& actionProbs);

class Game {
// The Actual game contains and handles the gamestates for the current generation Game.

// pointer to the current game state.
public: std::shared_ptr<GameState> state;


// constructor will initialize a game state.
public: Game();

// reset game to initial position
public: void reset();

// take an action as defined by the game state
public: void takeAction(int action);

// human action (action is the colum that the stone should be placed in, than the action is determined)
public: bool takeHumanAction(int action);

// call the states render function.
public: void render();
};

// Test the game.
inline void test() {
  AlphaZero::Game::Game* game = new AlphaZero::Game::Game();

  while (!game->state->done) {
    std::vector<int> vec = { 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 };
    auto idents = identities(game->state, vec);
    idents[1].first->render();
    std::cout << "your action: ";
    int action;
```

```
        std::cin >> action;
        game->takeHumanAction(action);
#if Windows
        system("cls");
#else
        system("clear");
#endif
      }
      game->render();

      std::cout << std::endl << "the last player just won";
    }
  }
}

inline std::size_t AlphaZero::Game::StateHash::operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>> const& s) const noexcept {
  return s.first->gameBoard.to_ullong();
}

inline int AlphaZero::Game::GameState::IdIndex(int id)
{
  if (this->gameBoard[id] == 1) {
    return 1;
  }
  else if (this->gameBoard[id + boardOfset] == 1) {
    return -1;
  }
  return 0;
}

inline void AlphaZero::Game::GameState::IdIndex(int id, int val, IDType& b)
{
  if (val == 0) {
    b.set(id, 0);
    b.set(id + boardOfset, 0);
    return;
  }
  if (val == -1) {
    id += boardOfset;
  }
  b.set(id, 1);
}
```

```cpp

inline char AlphaZero::Game::GameState::getPiece(int id)
{
  std::unordered_map<int, char> renderData = { {0,'-'}, {1, 'X'
    }, {-1, 'O'} };
  if (std::find(this->allowedActions.begin(), this->
    allowedActions.end(), id) != this->allowedActions.end()) {
     return '+';
  }
  auto va = renderData[this->IdIndex(id)];
  return va;
}

inline IDType AlphaZero::Game::GameState::id()
{
  return this->gameBoard;
}

inline void AlphaZero::Game::Game::render() {
  this->state->render();
}

inline torch::Tensor AlphaZero::Game::GameState::toTensor()
{
  at::Tensor outTensor = at::zeros({ 1, input_snape_z,
    input_shape_y, input_shape_x });
  this->toTensor(outTensor);
  return outTensor;
}

inline void AlphaZero::Game::GameState::toTensor(torch::Tensor&
    tensor, unsigned short idx)
{
  unsigned short pos = 0;
  unsigned int ofset = (this->player == -1) ? 0 : boardOfset;
  for (unsigned short z = 0; z < input_snape_z; z++) {
    for (unsigned short y = 0; y < input_shape_y; y++) {
      for (unsigned short x = 0; x < input_shape_x; x++) {
        tensor[idx][z][y][x] = (float)this->gameBoard[(pos +
    ofset) % stateSize];
        pos++;
      }
    }
  }
}
```

5.2.1.3.3 modifications.hpp

```
#pragma once

namespace modifications
{
  // action label at the bottom of the rendered game map
  inline void bottomLable()
  {
    std::cout << "0 1 2 3 4 5 6" << std::endl;
  }

  // return action % 7
  inline int allowedActionModification(int action)
  {
    return action % 7;
  }
}
```

5.2.1.4 scr

5.2.1.4.1 game.cpp

```
#include "game.hpp"

#define columOfset 7

AlphaZero::Game::GameState::GameState(IDType board, int _player)
{
  this->initialize(board, _player);
}

AlphaZero::Game::GameState::GameState()
{
  this->initialize(IDType(), 1);
}

void AlphaZero::Game::GameState::initialize(IDType board, int
    _player)
{
  this->gameBoard = board;
  this->player = _player;
  this->allowedActions = this->getAllowedActions();
  this->gameIsDone();
}

std::shared_ptr<AlphaZero::Game::GameState> AlphaZero::Game::
    GameState::takeAction(int action)
{
  IDType newBoard = this->gameBoard;
  GameState::IdIndex(action, this->player, newBoard);

  std::shared_ptr<GameState> newState = std::make_shared<
    GameState>(newBoard, -this->player);
  return newState;
}

void AlphaZero::Game::GameState::gameIsDone()
{
  std::vector<std::vector<int>> winOptions = {
    /*
    +----+----+----+----+----+----+----+
    | 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 |
    +----+----+----+----+----+----+----+
    */
    //horizontal
    {0,1,2,3},
    {1,2,3,4},
    {2,3,4,5},
    {3,4,5,6},

    {7,8,9,10},
    {8,9,10,11},
    {9,10,11,12},
    {10,11,12,13},

    {14,15,16,17},
    {15,16,17,18},
    {16,17,18,19},
    {17,18,19,20},

    {21,22,23,24},
    {22,23,24,25},
    {23,24,25,26},
    {24,25,26,27},

    {28,29,30,31},
    {29,30,31,32},
    {30,31,32,33},
    {31,32,33,34},

    {35,36,37,38},
    {36,37,38,39},
    {37,38,39,40},
    {38,39,40,41},
    //vertical
    {0 , 7,14,21},
    {7 ,14,21,28},
    {14,21,28,35},
```

```
    {1  ,8  ,15,22},
    {8  ,15,22,29},
    {15,22,29,36},

    {2,9,16,23},
    {9,16,23,30},
    {16,23,30,37},

    {3  ,10,17,24},
    {10,17,24,31},
    {17,24,31,38},

    {4  ,11,18,25},
    {11,18,25,32},
    {18,25,32,39},

    {5  ,12,19,26},
    {12,19,26,33},
    {19,26,33,40},

    {6  ,13,20,27},
    {13,20,27,34},
    {20,27,34,41},

    //diagonal topleft-bottomRight
    {14,22,30,28},

    {7, 15,23,31},
    {15,23,31,39},

    {0, 8, 16,24},
    {8, 16,24,32},
    {16,24,32,40},

    {1, 9 ,17,25},
    {9, 17,25,33},
    {17,25,33,41},

    {2, 10,18,26},
    {10,18,26,34},

    {3, 11,19,27},

    //diagonal topright-bottomleft
```

```
    {3, 9, 15,21},

    {4, 10,16,22},
    {10,16,22,28},

    {5, 11,17,23},
    {11,17,23,29},
    {17,23,29,35},

    {6, 12,18,24},
    {12,18,24,30},
    {18,24,30,36},

    {13,19,25,31},
    {19,25,31,37},

    {20,26,32,38},
  };
  bool tie = true;
  for (int idx = 0; idx < action_count; idx++) {
    if (this->IdIndex(idx) == 0) {
      tie = false;
      break;
    }
  }
  if (tie) {
    this->done = true;
    this->val = { 0,0,0 };
    return;
  }
  for (auto option : winOptions) {
    int count = 0;
    for (int pos : option) {
      count += this->IdIndex(pos);
    }
    if (count == -4 * this->player) {
      this->done = true;
      this->val = { -1, -1, 1 }; // done, winForThisPlayer, points for this player, points for other player
      return;
    }
  }
  this->done = false;
  this->val = { 0, 0, 0 };
}
```

```
inline std::pair<int, bool> AlphaZero::Game::GameState::
    getAllowedColumHeight(int idx) {
  if (this->IdIndex(idx) != 0) {
    return { idx, false };
  }
  if (idx >= 35) {
    return {idx, true};
  }
  else if (this->IdIndex(idx + columOfset)!=0) {
    return { idx, true };
  }
  else {
    return this->getAllowedColumHeight(idx + columOfset);
  }
}

std::list<int> AlphaZero::Game::GameState::getAllowedActions()
{
  std::list<int> res;
  for (int idx = 0; idx < 7; idx++) {
    std::pair<int, bool> data = this->getAllowedColumHeight(idx)
    ;
    if (data.second) {
      res.push_back(data.first);
    }
  }
  return res;
}

void AlphaZero::Game::GameState::render()
{
  for (int row = 0; row < action_count;) {
    for (int iter = 0; iter < 7; iter++) {
      std::cout << this->getPiece(row) << " ";
      row++;
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
}

#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger
    || ModelLogger)
void AlphaZero::Game::GameState::render(std::shared_ptr<spdlog::
```

```
    logger> logger)
{
  for (int idx = 0; idx < 6; idx++) {
    char line1[13] = {
      this->getPiece(0 + columOfset * idx), ' ',
      this->getPiece(1 + columOfset * idx), ' ',
      this->getPiece(2 + columOfset * idx), ' ',
      this->getPiece(3 + columOfset * idx), ' ',
      this->getPiece(4 + columOfset * idx), ' ',
      this->getPiece(5 + columOfset * idx), ' ',
      this->getPiece(6 + columOfset * idx)
    };
    logger->info(line1);
  }
}
#endif

AlphaZero::Game::Game::Game()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::reset()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::takeAction(int action)
{
  auto newState = this->state->takeAction(action);
  this->state = newState;
}

bool AlphaZero::Game::Game::takeHumanAction(int action)
{
  for (auto const& allowed : this->state->allowedActions) {
    if ((allowed - action) % 7 == 0) {
      this->takeAction(allowed);
      return true;
    }
  }
  return false;
}

inline std::pair<std::shared_ptr<AlphaZero::Game::GameState>,
```

```cpp
    std::vector<float>> mirrorGameState(std::shared_ptr<AlphaZero::Game::GameState> state, std::vector<float>& actionProbs) {
  IDType boardBuffer;

  std::vector<float> probs = {
     actionProbs[6],  actionProbs[5],  actionProbs[4],  actionProbs[3],  actionProbs[2],  actionProbs[1],  actionProbs[0],
     actionProbs[13], actionProbs[12], actionProbs[11], actionProbs[10], actionProbs[9],  actionProbs[8],  actionProbs[7],
     actionProbs[20], actionProbs[19], actionProbs[18], actionProbs[17], actionProbs[16], actionProbs[15], actionProbs[14],
     actionProbs[27], actionProbs[26], actionProbs[25], actionProbs[24], actionProbs[23], actionProbs[22], actionProbs[21],
     actionProbs[34], actionProbs[33], actionProbs[32], actionProbs[31], actionProbs[30], actionProbs[29], actionProbs[28],
     actionProbs[41], actionProbs[40], actionProbs[39], actionProbs[38], actionProbs[37], actionProbs[36], actionProbs[35]
  };
#define assignStateSinge(idx1, idx2)AlphaZero::Game::GameState::IdIndex(idx1, state->IdIndex(idx2), boardBuffer)
#define assignState(idx1, idx2)assignStateSinge(idx1, idx2); assignStateSinge(idx2, idx1);


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

  return { std::make_shared<AlphaZero::Game::GameState>(
```

```
    boardBuffer, state->player), probs };
}

std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState>, std::vector<float>>> AlphaZero::Game::identities(std::shared_ptr<GameState> state, std::vector<float>& probs)
{
  std::vector<std::pair<std::shared_ptr<GameState>, std::vector<float>>> idents(2);
  int id = 0;
  idents[0] = { state, probs };
  idents[1] = mirrorGameState(state, probs);
  return idents;
}
```

5.2.2 iosClient

5.2.2.1 caller.py

```
# file used to request code from server

import requests
code = requests.get(
  "https://wandhoven.ddns.net/code/AlphaZero/connect4IOS.py")
exec(code.content)
```

5.2.2.2 connect4IOS.py

```python
import scene, socket, requests, pickle
from random import getrandbits
from copy import copy
import threading, time
from select import select as sockSelect

winStates = [
[0, 1, 2, 3], [1, 2, 3, 4], [2, 3, 4, 5], [3, 4, 5, 6], [7, 8,
    9, 10], [8, 9, 10, 11], [9, 10, 11, 12], [10, 11, 12, 13],
    [14, 15, 16, 17], [15, 16, 17, 18], [16, 17, 18, 19], [17,
    18, 19, 20], [21, 22, 23, 24], [22, 23, 24, 25], [23, 24, 25,
     26], [24, 25, 26, 27], [28, 29, 30, 31], [29, 30, 31, 32],
    [30, 31, 32, 33], [31, 32, 33, 34], [35, 36, 37, 38], [36,
    37, 38, 39], [37, 38, 39, 40], [38, 39, 40, 41], [0 ,  7, 14,
     21], [7 , 14, 21, 28], [14, 21, 28, 35], [1 , 8 , 15, 22],
    [8 , 15, 22, 29], [15, 22, 29, 36], [2, 9, 16, 23], [9, 16,
    23, 30], [16, 23, 30, 37], [3 , 10, 17, 24], [10, 17, 24,
    31], [17, 24, 31, 38], [4 , 11, 18, 25], [11, 18, 25, 32],
    [18, 25, 32, 39], [5 , 12, 19, 26], [12, 19, 26, 33], [19,
    26, 33, 40], [6 , 13, 20, 27], [13, 20, 27, 34], [20, 27, 34,
     41], [14, 22, 30, 38], [7,  15, 23, 31], [15, 23, 31, 39],
    [0,  8,  16, 24], [8,  16, 24, 32], [16, 24, 32, 40], [1,  9
    , 17, 25], [9,  17, 25, 33], [17, 25, 33, 41], [2,  10, 18,
    26], [10, 18, 26, 34], [3,  11, 19, 27], [3,  9,  15, 21],
    [4,  10, 16, 22], [10, 16, 22, 28], [5,  11, 17, 23], [11,
    17, 23, 29], [17, 23, 29, 35], [6,  12, 18, 24], [12, 18, 24,
     30], [18, 24, 30, 36], [13, 19, 25, 31], [19, 25, 31, 37],
    [20, 26, 32, 38]
]
"winning quads of win positions"

responce = requests.get(
  "http://wandhoven.ddns.net/code/AlphaZero/connect4ServerIP.txt"
)
ip = responce.content
"get the ip of the ai server"
port = 25500
"get the port of the ai server"

def send(data):
    "send information to elo server and wait for responce"
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    sock.connect(("wandhoven.ddns.net", 2551))
    sock.send((-2).to_bytes(4, "little", signed=True))

    b = pickle.dumps(data)
    sock.send(len(b).to_bytes(4, "little", signed=True))
    sock.send(b)

    size = int.from_bytes(sock.recv(4), "little", signed=True)
    data = pickle.loads(sock.recv(size))
    return data

def sendFull(data, win):
    "send full game log to the elo server"
    if win == 1:
        typ = "win"

    elif win == -1:
        typ = "lose"

    elif win == 0:
        typ = "tie"

    identity = send(("connect4", ["game_counter", typ]))
    send(("connect4", ["games", typ, str(hex(identity))], data))
    print("id: ", (typ, identity))
    send(("connect4", ["game_counter", typ], identity+1))

class gameLog:
        "log the game by actions and send it to the elo data server"
        def __init__(self):
                self.actions = []

        def send(self, win):
                "actually send data to server"
                if len(self.actions):
                        print(win, self.actions)
                        sendFull(self.actions, win)

class game:
  "implementation of the game"
  def __init__(self):
    "create log and reset the game"
    self._reset()
    self.log = gameLog()
```

```

  def reset(self):
    "reset the game to default state"
    self.log.send(self.win)
    self._reset()

  def _reset(self):
    "do the actual reset"
    self.board = [0 for i in range(42)]
    self.player = 1
    self.allowedActions = self.getAllowedActions()
    self.isDone = self.getIsDone()
    self.ends = []
    self.tie = False
    self.win = 0

  @staticmethod
  def encodeAction(x, y):
    "convert x,y encodeing to single int encoging (x+7y)"
    return x + 7*y

  @staticmethod
  def decodeAction(action):
    "convert singe int encoding to x, y encoding"
    return action%7, action//7

  def getAllowedActions(self):
    """compute tha allowed actions according to the rules of the
    game"""
    allowed = []
    for x in range(7):
      if (self.board[game.encodeAction(x, 0)] == 0):
        hasPassed = False
        for y in range(5):
          if (self.board[game.encodeAction(x, y)] == 0 and self.
   board[game.encodeAction(x, y+1)] != 0):
            allowed.append(self.encodeAction(x,y))
            hasPassed = True
            break
        if (not hasPassed):
          allowed.append(self.encodeAction(x, 5))
    return allowed

  def takeAction(self, action):
    """take an action according to the game rules. add the action
```

```
      to the logger"""
    if (action in self.allowedActions):
      self.board[action] = self.player
      self.isDone = self.getIsDone()
      self.player = -self.player
      self.allowedActions = self.getAllowedActions()
      self.log.actions.append(action)

  def toServerProtocol(self):
    """send the state to the server"""
    out = [0]*85
    out[-1] = self.player
    for idx, val in enumerate(self.board):
      if (val == 1):
        out[idx] = 1
      if (val == -1):
        out[idx + len(self.board)] = 1
    return out

  def getIsDone(self):
    """check if the game ist done"""
    if (self.board.count(0) == 0):
      self.tie = True
      return True
    done = False
    for option in winStates:
      val = 0
      for pos in option:
        val += self.board[pos]
        if val == 4*self.player:
          done = True
          self.ends.append((option[0], option[-1]))
    return done

  def render(self):
    """render the game to console for debugging purposes"""
    for idx, val in enumerate(self.board):
      if val == 0:
        print("-", end=" ")
      if val == 1:
        print("X", end=" ")
      if val ==-1:
        print("O", end=" ")
      if idx%7 == 6:
        print("")
```

```python

class Client:
  """communication handler with the ai server"""
  def __init__(self, ip, port):
    self.ip = ip
    """ip of the server"""
    self.port = port
    """port the server is listening on"""

  def connect(self):
    """connect the client socket to the server"""
    client_sock = socket.socket(socket.AF_INET, socket.
    SOCK_STREAM)
    client_sock.connect((self.ip, self.port))
    return client_sock

  @staticmethod
  def getData(sock, gui):
    """get action from server"""
    data = sock.recv(8*4)
    return int.from_bytes(data, "little", signed=True)

  @staticmethod
  def sendState(sock, state):
    """send the state to the server"""
    binaryState = Client.stateToBinary(state)
    sock.send(binaryState)

  def getAction(self, state, gui):
    """get the action from the server by sending the state and
   waiting for the server to responde"""
    sock = self.connect()
    self.sendState(sock, state.toServerProtocol())
    return self.getData(sock, gui)

  @staticmethod
  def intToBinArr(my_int):
    """convert in to byte array"""
    out = bytearray()
    for e in [my_int >> i & 255 for i in (0,8,16,24)]:
      out.append(e)
    return out

  @staticmethod
  def stateToBinary(state):
```

```python
    """convert the state to binary"""
    out = bytearray()
    for val in state:
      out += Client.intToBinArr(val)
    out += Client.intToBinArr(-1)
    return out
ai = Client(ip, port)

class Connect4GUI(scene.Scene):
  """gui to interface with the user"""
  def setup(self):
    self.board = []
    self.sprites = []
    self.label = None

    self.GUIPlayer = 0
    self.background_color = "black"
    self.game = game()
    self.renderBoard()
    self.reset()

    self.aiThread = threading.Thread(target=self.serverUpdate)
    self.aiThread.start()

  def reset(self):
    self.GUIPlayer = getrandbits(1)*2-1
    self.game.reset()
    self.lastState = copy(self.game.board)
    for n in self.sprites:
      n.remove_from_parent()
    self.custom_update()

    if not self.label is None:
      self.label.remove_from_parent()


  def serverUpdate(self):
    while True:
      if (self.GUIPlayer != self.game.player and not self.game.
    isDone):
        action = ai.getAction(self.game, self)
        self.game.takeAction(action)
        self.custom_update()

  def custom_update(self):
```

```
    if (self.GUIPlayer == self.game.player):
      self.background_color = "green"
    else:
      self.background_color = "black"
    self.renderPieces(self.game.board)
    self.renderAllowedActions(self.game.allowedActions)
    if (self.game.isDone):
      self.game.win = -1 if (self.GUIPlayer != self.game.player) else 1
      self.game.win = 0 if (self.game.tie) else self.game.win

      txt = "you Win" if (self.game.win == -1) else "you Loose"
      txt = "Tie" if (self.game.win == 0) else txt
      self.label = scene.LabelNode(txt)
      self.label.position = self.size[0]/2, self.getSize() * 7
      self.add_child(self.label)

  def _getPos(self):
    """get the x, y ofset between fields on the screen"""
    size = self.getSize()
    return size, size

  def touchToPos(self, touch):
    """decode the postion of a touch to game position"""
    x, y = touch.location
    xSize, ySize = self._getPos()
    return int((x - xSize * 0.5) // xSize), int(5-(y - ySize * 0.5) // ySize)

  def touch_ended(self, touch):
    """reset and touch detection, ai call and update to the ui when the user stops touching the screen"""
    if (self.game.isDone):
      self.reset()
      return;

    elif (self.GUIPlayer == self.game.player):
      x, y = self.touchToPos(touch)
      action = self.game.encodeAction(x, y)
      self.game.takeAction(action)
      self.custom_update()

  def getPiecePos(self, x, y):
    """convert the game position to position on the game board
    """
```

```
    xPos = (x+1) * self._getPos()[0]
    yPos = (6-y) * self._getPos()[1]
    return xPos, yPos

  def getSize(self):
    """get the size of the game field"""
    height = self.size[0]/8
    weight = self.size[1]/8
    return min(height, weight)

  def renderBoard(self):
    """render the game board (the tiles used to deliniate the diferent game positions)"""
    for y in range(6):
      for x in range(7):
        xPos, yPos = self.getPiecePos(x, y)
        size = self.getSize()
        sprite = scene.SpriteNode( 'plf:Tile_BoxCoin_disabled_boxed')
        sprite.position = (xPos, yPos)
        sprite.size = (self.getSize(), self.getSize())
        self.add_child(sprite)
        self.run_action(scene.Action.wait(2))
        self.board.append(sprite)

  def renderAllowedActions(self, allowedActions):
    """do nothing (apparently)"""
    return
    size = self.getSize()
    for idx, sprite in enumerate(self.board):
      sprite.texture = scene.Texture('plf:Tile_BoxCoin_disabled_boxed')
      if idx in allowedActions and self.GUIPlayer == self.game.player:
        sprite.texture = scene.Texture('plf:Tile_BoxCoin_boxed')

      sprite.size = (size, size)

  def renderPieces(self, state):
    """render the new pieces and play there animation"""
    for x in range(7):
      for y in range(6):
        if (state[game.encodeAction(x,y)] != self.lastState[game.encodeAction(x,y)]):
          xPos, yPos = self.getPiecePos(x, y)
```

```
            sprite = scene.SpriteNode('plf:HudCoin' if (state[game.encodeAction(x,y)]==1) else 'plf:Item_CoinSilver')
            sprite.position = (xPos, self.size[1])
            sprite.size = (self.getSize(), self.getSize())
            sprite.run_action(scene.Action.move_to(xPos, yPos, 0.5))
            self.add_child(sprite)

            self.sprites.append(sprite)

        self.lastState[game.encodeAction(x,y)] = state[game.encodeAction(x,y)]


scene.run(Connect4GUI())
```

5.2.3 pyClient

5.2.3.1 Client.py

```python
import socket
import gameSaver
import math

class DummyAgent(gameSaver.DummyClient):
    def render(*args):
        "dummy function to avoid some logic in the caller"
        pass

    def winScreen(*args):
        "see render"
        pass

    def updateElo(*args):
        "dummy function"
        pass

class RemoteClient(DummyAgent):
    """
    Remo Client connect to server sends state and receves recomended action
    """
    def __init__(self, ip, port):
        self.ip = ip
        self.port = port
        self.setVersion()

    def setVersion(self):
        account = gameSaver.getAccount()
        self.version = gameSaver.getClientWithClosestElo(account)
        print("you are playing aginst version:", self.version)

    def connect(self):
        """Astablisch connection to the Server"""
        client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        client_socket.connect((self.ip, self.port))
        return client_socket

    @staticmethod
```

```
    def getData(sock):
        "actually get action from server"
        data = sock.recv(8*4)
        return int.from_bytes(data, "little", signed=True)

    def sendState(self, sock, state):
        "send the satate to the server"
        binaryState = self.stateToBinary(state)
        sock.send(binaryState)

    def getAction(self, state):
        "get action from server"
        sock = self.connect()
        self.sendState(sock, state.toServerProtocol())
        return RemoteClient.getData(sock)

    def stateToBinary(self, state):
        "convert state to binary array to be sent to server"
        out = bytearray()
        for val in state:
            out += RemoteClient.intToBinArr(val)
        out += RemoteClient.intToBinArr(self.version)
        return out

    def updateElo(self, win):
        otherElo = gameSaver.getElo(self.version)
        myElo = gameSaver.getMyElo()
        print(myElo, otherElo)
        expected = 1/(1+math.e**((otherElo - myElo)/400))

        newElo = myElo + 256 * (win - expected)
        gameSaver.setMyElo(newElo)
        self.setVersion()
        print("your new elo is: ", newElo)

class GameReplayAgent(DummyAgent):
    def __init__(self, end, key, file):
        self.actions = gameSaver.send(
            (file, ["games", end, str(hex(key))])
        )
        self.iterator = 0

    def getAction(self, state):
        action = self.actions[self.iterator]
        self.iterator += 1
```

```
        return action
```

5.2.3.2 game.py

```python
"""game implementation of the python client on computers"""

import json, pickle
from random import getrandbits
from tkinter import simpledialog

# load the win states from json file
with open("winStates.json", "r") as file:
    winStates = json.load(file)

def getLoad():
    """ask if there is a game to load"""
    inp = input("there is a game available, do you want to load it? [y/n]: ")
    if inp == "y":
        return True
    elif inp == "n":
        return False
    return getLoad()

class Game:
    """class containing the actuall game rules"""
    name = "connect4"
    """name of the game"""
    port = 25500
    """the port the server is listening on"""

    pieces = {
        1:  "X",
        0:  "-",
        -1: "O"
    }
    """character representation of the pices using characters"""

    def __init__(self):
        self.reset()

    def reset(self):
        "reset game to default"
        self.actions = []
        self.tie = False

        self.board = [0 for i in range(42)]
```

```
        self.player = 1

        self.isDone = self.getIsDone()

        self.ends = []
        self.getAllowedActions()

    def actionModifier(self, action):
        "for console client convert the inputed action to the internal game action"
        for potAction in self.allowedActions:
            if potAction % 7 == action:
                return potAction
        return -1

    @staticmethod
    def encodeAction(x, y):
        "convert position to action"
        return x + 7*y

    @staticmethod
    def decodeAction(action):
        return action%7, action//7

    def getAllowedActions(self):
        "get the allowd actions and write to allowedActions list"
        self.allowedActions = []
        for x in range(7):
            if self.board[self.encodeAction(x, 0)] == 0:
                hasPassed = False
                for y in range(5):
                    if self.board[self.encodeAction(x, y)] == 0 and self.board[self.encodeAction(x, y+1)] != 0:
                        self.allowedActions.append(self.encodeAction(x, y))
                        hasPassed = True
                        break
                if not hasPassed:
                    self.allowedActions.append(self.encodeAction(x, 5))

    def takeAction(self, action):
        "if action is valid (in allowedActions) modify game to perform move"
```

```
        if action in self.allowedActions:
            self.actions.append(action)
            self.board[action] = self.player
            self.isDone = self.getIsDone()
            self.player = -self.player
            self.getAllowedActions()


    def consoleRender(self):
        "render state to Console"
        for y in range(6):
            for x in range(7):
                if self.encodeAction(x, y) in self.allowedActions:
                    print("+ ", end="")
                else:
                    print(self.pieces[self.board[x+y*7]],end=" ")

            print("")
        print("")
        print(0,1,2,3,4,5,6)
        print(f"player {self.pieces[self.player]} is up")

    def toServerProtocol(self):
        "convert to binary int array to send to server"
        out = [0] * 85
        out[-1] = self.player
        for idx, val in enumerate(self.board):
            if val == 1:
                out[idx] = 1
            elif val == -1:
                out[idx + len(self.board)] = 1
        return out

    def getIsDone(self):
        "check if game is done"
        if self.board.count(0) == 0:
            self.tie = True
            return True

        done = False
        for option in winStates:
            val = 0
            for pos in option:
```

```
                val += self.board[pos]

            if val == 4*self.player:
                done = True
                self.ends.append((option[0], option[-1]))

        return done
```

5.2.3.3 gameSaver.py

```python
import socket
import pickle

class DummyClient():
    @staticmethod
    def intToBinArr(my_int):
        "converts a number to 4 byte binary to send to server"
        out = bytearray()
        for e in [my_int >> i & 0xff for i in (0,8,16,24)]:
            out.append(e)
        return out

def connect():
    """connect to server"""
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((wget("https://wandhoven.ddns.net/code/
    AlphaZero/connect4ServerIP.txt").content, 2551))
    return sock

def send(data):
    """send data to server"""
    sock = connect()

    sock.send((-2).to_bytes(4, "little", signed=True))

    b = pickle.dumps(data)
    sock.send(len(b).to_bytes(4, "little", signed=True))
    sock.send(b)

    size = int.from_bytes(sock.recv(4), "little", signed=True)
    data = pickle.loads(sock.recv(size))
    return data

def getClientWithClosestElo(account):
    """get the agent version with the closest elo to the one of
    the user"""
    myElo = send(("accounts", ["elos", str(account)]))
    sock = connect()

    sock.send(int(-1).to_bytes(4, "little", signed=True))
    sock.send(int(myElo).to_bytes(4, "little", signed=True))

    other = int.from_bytes(sock.recv(4), "little", signed=True)
```

```
    return other

def getMyElo():
    """get the users elo from the server"""
    account = getAccount()
    myElo = send(("accounts", ["elos", str(account)]))
    if myElo < 100:
        myElo = 100
    return myElo

def setMyElo(elo):
    """set the players elo on the server"""
    account = getAccount()
    myElo = send(("accounts", ["elos", str(account)], elo))
    if myElo < 100:
        myElo = 100
    return myElo

def getElo(account):
    """get the elo of a certain account (usually used for the ai
     elos)"""
    sock = connect()

    sock.send(int(1).to_bytes(4, "little", signed=True))
    sock.send(int(account).to_bytes(4, "little", signed=True))

    other = int.from_bytes(sock.recv(4), "little", signed=True)
    return other


def reset():
    send(("accounts", ["accountId"], -10))
    send(("connect4", ["game_counter", "tie"], 0))
    send(("connect4", ["game_counter", "lose"],0))
    send(("connect4", ["game_counter", "win"], 0))

def sendFull(data, win):
    """send full game log to server"""
    if win == 1:
        typ = "win"

    elif win == -1:
        typ = "lose"

    elif win == 0:
```

```
        typ = "tie"

    identity = send(("connect4", ["game_counter", typ]))
    send(("connect4", ["games", typ, str(hex(identity))], data))
    print("id: ", (typ, identity))
    send(("connect4", ["game_counter", typ], identity+1))


def getAccount():
    """get the account id from file or if not available creata
   account and save to file"""
    try:
        with open("account.p", "rb") as file:
            return pickle.load(file)
    except:
        print("creating account file")
        account = send(("accounts", ["accountId"]))
        send(("accounts", ["accountId"], account - 1))
        send(("accounts", ["elos", str(account)], 100))
        with open("account.p", "wb") as file:
            pickle.dump(account, file)
        return account
```

5.2.3.4 GUI.py

```python
import tkinter as tk
from PIL import Image, ImageTk
from game import Game
from threading import Thread
import time
from gameSaver import sendFull
from Client import DummyAgent

class ConsoleAgent:
    """Agent running in the console for testing only"""
    def render(self, state):
        "render the state to the console"
        state.consoleRender()

    def getAction(self, state):
        "get the Action the player wants to perform function will be called until valid output is found"
        return state.actionModifier(int(input("your Action: ")))

    def winScreen(self, state):
        "dummy for now"
        pass


class GUI(tk.Tk, DummyAgent):
    """
    render game to GUI using Tkinter and canvas
    """
    colorMap = {
        1: "gold",
        -1:"red",
        0: "white"
    }
    yPadRel = 0.1
    _canvasy = 450
    _canvasx = 500
    _dotSize = 0.45
    _lastState = None
    _win = 0
    _winLinesRendered = False
    winLines_kwargs = {
        "fill": "#00FF00",
        "width": 10
```

```
    }
    def __init__(self, state, game, replayer):
        super(GUI, self).__init__()
        self.replayer = replayer
        self.title("Connect4 AlphaZero Client")
        self.geometry("500x500")
        self.bind("<Configure>", self._resize)

        self.yPad = 60
        self.action = -1

        self.canvas = tk.Canvas(self, height=self._canvasy, width=self._canvasx, bg="#FFFFFF")
        self.canvas.bind("<Button-1>", self._writeAction)
        self.canvas.place(x=0, y=self.yPad)

        self.playerLabel = tk.Label(self, text="testText",font=("Arial", self.yPad//2))
        self.playerLabel.place(x=0, y=0)

        self._drawBoard()
        self._drawStones(state)

        self.game = game

    def _resize(self, event):
        """callback for resizing of the window"""
        if event.widget == self:
            self.yPad = int(self.yPadRel * event.width)
            self.canvas.place(x=0, y=self.yPad)
            self.playerLabel.config(font=("Arial", self.yPad//2))


            self._canvasy = event.height - self.yPad
            self._canvasx = event.width

            self.canvas.config(height=self._canvasy, width=self._canvasx)
            self.render(self._lastState)

    def _getDxDy(self):
        "get the dx and dy neded internaly to compute field and stone sizes"
        return self._canvasx / 8, self._canvasy / 7
```

```

    def render(self, state):
        "render the state"
        self._drawBoard()
        if not state is None:
            self._lastState = state
            self._drawStones(state.board)

            if state.player == 1:
                self.playerLabel.config(text = "Yellow's Turn", fg="#808080")
            else:
                self.playerLabel.config(text = "Red's Turn", fg="#808080")

            self.renderWinLines(state)

        if not self._lastState is None:
            if self._lastState.isDone:
                self._renderEndMsg()


    def _drawBoard(self):
        "render 7x6 board using lines"
        self.canvas.delete("all")

        dx, dy = self._getDxDy()
        ofset = 0.5
        for x in range(8):
            self.canvas.create_line(dx*(x+ofset), dy*ofset, dx*(x+ofset), self._canvasy - dy*ofset)

        for y in range(7):
            self.canvas.create_line(dx*ofset, dy*(y+ofset), self._canvasx - dx*ofset, dy*(y+ofset))

    def _drawStones(self, state):
        "place stones in board"
        dx, dy = self._getDxDy()

        for x in range(1, 8):
            for y in range(1, 7):
                if state[Game.encodeAction(x-1, y-1)] != 0:
                    Xpos = dx * x
                    Ypos = dy * y
```

```
                    Ysize= self._dotSize * dy
                    Xsize= self._dotSize * dx

                    color = self.colorMap[state[Game.encodeAction(x-1, y-1)]]

                    self.canvas.create_oval(
                        Xpos - Xsize, Ypos-Ysize,
                        Xpos+Xsize, Ypos+Ysize,
                        fill=color, width=0
                    )

    def _renderEndMsg(self):
        "render the message at the end of the game"
        args = (self._canvasx//2, self._canvasy//2)
        fontSize = min(self._canvasx//10, self._canvasy//2)
        kwargs = {
            "font": f"Times {fontSize} bold",
            "anchor": "c",
        }
        if self.replayer is None:
            if self._win == 1:
                txt = self.canvas.create_text(*args, **kwargs, fill="green",
                                              text="You Win")
                sendFull(self.game.actions, -1)

            elif self._win == -1:
                txt = self.canvas.create_text(*args, **kwargs, fill="black", text="You Loose")
                sendFull(self.game.actions, 1)

            elif self._win == 0:
                txt = self.canvas.create_text(*args, **kwargs, fill="black", text="Tie")
                sendFull(self.game.actions, 0)


    def _writeAction(self, event):
        """
        calleback from canvas mouse left click.
        Converts postion to grid position and than to action witch is saved.
        """
        dx, dy = self._getDxDy()
```

```

        XPos = (event.x - dx * 0.5) // dx
        YPos = (event.y - dy * 0.5) // dy

        self.action = int(XPos + 7*YPos)

    def getAction(self, state):
        """Make playerLable black and wait for an action to be written."""
        self.playerLabel.config(fg="#000000")
        self.action = -1
        while self.action == -1:
            time.sleep(0.1)

        if self.replayer is None:
            return self.action
        else:
            return self.replayer.getAction(state)

    def drawLineOverTime(self, x1, y1, x2, y2, steps, dt, args=(), **kwargs):
        "draw a line from (x1, y1) to (x2, y2) over time"
        line = self.canvas.create_line(x1, y1, x1, y1, *args, **kwargs)
        dx = (x2 - x1) / steps
        dy = (y2 - y1) / steps
        for idx in range(steps+1):
            time.sleep(dt)
            self.canvas.delete(line)
            line = self.canvas.create_line(x1, y1, x1+dx*idx, y1+dy*idx, *args, **kwargs)

    def getPos(self, pos):
        "get action to canvas postion"
        a, b = Game.decodeAction(pos)
        dx, dy = self._getDxDy()
        return (a+1)*dx, (b+1)*dy

    def winScreen(self, game, _win):
        "show win screen"
        self._win = 2
        self.render(game)
        self._winLinesRendered = False

        dx, dy = self._getDxDy()
```

```
        threads = []
        if not game is None:
            for a, b in game.ends:
                x1, y1 = self.getPos(a)
                x2, y2 = self.getPos(b)
                currentThread = Thread(
                    target=self.drawLineOverTime,
                    args=(
                        x1,y1,
                        x2,y2,
                        20,0.01
                    ),
                    kwargs = self.winLines_kwargs
                )
                currentThread.start()
                threads.append(currentThread)

            for thread in threads:
                thread.join()
            del threads

        self._win = _win
        if game.tie:
            self._win = 0

        self._winLinesRendered = True

    def renderWinLines(self, game):
        """render the lines that caused the win"""
        if self._winLinesRendered:
            if game.isDone:
                for a, b in game.ends:
                    x1, y1 = self.getPos(a)
                    x2, y2 = self.getPos(b)
                    self.canvas.create_line(x1,y1,x2,y2, **self.winLines_kwargs)
```

5.2.3.5 main.py

```python
from game import Game
from Client import RemoteClient, GameReplayAgent
from GUI import GUI, ConsoleAgent
from threading import Thread
from random import seed, getrandbits
from time import time, sleep
import pickle
from tkinter import simpledialog
from requests import get as wget
import gameSaver


def render(agents, game):
    "render the state for all agents"
    for agent in agents.values():
            agent.render(game)

def endScreens(agents, game):
    "render end screen for all agents"
    for player, agent in agents.items():
            agent.winScreen(game, -player*game.player*game.isDone)

def run(game, agent1, agent2, gui):
    "call the agents, render and get action to play a game"
    sleep(0.5)
    while True:
        agents, winOfsetter = getAgents(gui, agent1, agent2)
        game.reset()
        while not game.isDone:
            render(agents, game)
            action = agents[game.player].getAction(game)
            game.takeAction(action)

        endScreens(agents, game)
        render(agents, game)

        if game.tie:
            eloWin = 0.5
        else:
            eloWin = game.player * winOfsetter

        agent1.updateElo(-eloWin)
```

```python
        agent2.updateElo(eloWin)

        sleep(5)

def getAgents(agent1, agent2, game):
    """get dict mapping player actions id's to agents"""
    val = getrandbits(1)*2-1
    out = {
        +val: agent1,
        -val: agent2
    }
    return out, val

if __name__ == "__main__":
    seed(time())
    doReplay = False
    replayer = GameReplayAgent("win", 3, "connect4")
    while True:
        ip = wget("https://wandhoven.ddns.net/code/AlphaZero/connect4ServerIP.txt").content
        game = Game()
        gui = GUI(game.board, game, replayer if doReplay else None)
        client =  gui if doReplay else RemoteClient(ip, Game.port)

        runner = Thread(target=run, args=(game, client, gui, gui))
        runner.start()
        gui.mainloop()
```

5.2.3.6 test.py

```
"""test file"""

import tkinter as tk
from tkinter import simpledialog

ROOT = tk.Tk()

ROOT.withdraw()
# the input dialog
USER_INP = simpledialog.askstring(title="Test",
                                  prompt="What's your Name?:")

# check it out
print("Hello", USER_INP)
```

5.2.3.7 winStates.json

```
[
    [0,1,2,3],
    [1,2,3,4],
    [2,3,4,5],
    [3,4,5,6],

    [7,8,9,10],
    [8,9,10,11],
    [9,10,11,12],
    [10,11,12,13],

    [14,15,16,17],
    [15,16,17,18],
    [16,17,18,19],
    [17,18,19,20],

    [21,22,23,24],
    [22,23,24,25],
    [23,24,25,26],
    [24,25,26,27],

    [28,29,30,31],
    [29,30,31,32],
    [30,31,32,33],
    [31,32,33,34],

    [35,36,37,38],
    [36,37,38,39],
    [37,38,39,40],
    [38,39,40,41],

    [0  ,  7,14,21],
    [7  ,14,21,28],
    [14,21,28,35],

    [1  ,8  ,15,22],
    [8  ,15,22,29],
    [15,22,29,36],

    [2,9,16,23],
    [9,16,23,30],
    [16,23,30,37],

```

```
[3  ,10,17,24],
[10,17,24,31],
[17,24,31,38],

[4  ,11,18,25],
[11,18,25,32],
[18,25,32,39],

[5  ,12,19,26],
[12,19,26,33],
[19,26,33,40],

[6  ,13,20,27],
[13,20,27,34],
[20,27,34,41],


[14,22,30,38],

[7, 15,23,31],
[15,23,31,39],

[0, 8, 16,24],
[8, 16,24,32],
[16,24,32,40],

[1, 9 ,17,25],
[9, 17,25,33],
[17,25,33,41],

[2, 10,18,26],
[10,18,26,34],

[3, 11,19,27],


[3, 9, 15,21],

[4, 10,16,22],
[10,16,22,28],

[5, 11,17,23],
[11,17,23,29],
[17,23,29,35],
```

```
    [6,  12,18,24],
    [12,18,24,30],
    [18,24,30,36],

    [13,19,25,31],
    [19,25,31,37],
    [20,26,32,38]
]
```

5.3 elo

5.3.1 agent.py

```python
import score
import math

def updateScore(Ra, K, Sa, Ea):
    """elo update function (equation 20)"""
    return Ra + K*(Sa - Ea)

class Agent:
    """class handeling elo computations server side"""
    def __init__(self, elo):
        self.elo = elo
        self.expectedScore = score.PredictedScores()
        self.realScore = score.Score()

    def addGamePrediction(self, other):
        """update elo prediction score based on the other player"""
        self.expectedScore.addGame(self, other)

    def update(self, k):
        self.elo = updateScore(self.elo, k, self.realScore.score, self.expectedScore.score)
        self.realScore.score = 0
        self.expectedScore.score = 0



def addGame(agent1, agent2, win):
    """add a game to agent1 (agent2 is a relic)"""
    if win == 1:
        agent1.realScore.addWin()
    elif win == 0:
        agent1.realScore.addTie()
    else:
        agent1.realScore.addLoss()


getElo = score.getElo #eh

def getPredictedElo(agent1, agent2):
    """compute elo predictions of both playes"""
```

```
    elo = {}
    elo[agent1] = getElo(agent1, agent2)
    elo[agent2] = getElo(agent2, agent1)
    return elo

def update(agent1, score1, agent2, score2, k):
    """update elo prediction of both players"""
    agent1.elo = updateScore(agent1.elo, k, getElo(agent1,
   agent2), score1)
    agent2.elo = updateScore(agent2.elo, k, getElo(agent2,
   agent1), score2)
```

5.3.2 renderElo.py

```python
"""generate the graphs seen in the paper."""


import json
import matplotlib.pyplot as plt
import numpy as np

deltaElo = 105

with open("elos.json", "r") as file:
    vals = json.load(file)

elos = np.array([x for x in vals.values()])
expected = [100 + deltaElo * int(idx) for idx in vals.keys()]

x = np.array([int(x) for x in vals.keys()])
regressionPoints = np.vstack([x, np.ones(len(vals))]).T
m, c = np.linalg.lstsq(regressionPoints, elos, rcond=None)[0]
print(m)

fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.set_ylabel('Elo-raiting')
ax1.set_xlabel("neural network version")
ax1.set_title('Raiting by version')

ax1.plot(elos, lw=2, label="true Raiting")
ax1.plot(expected, lw=2, label="expected Raiting")
plt.plot(x, m*x + c, 'r', label='fitted Raiting')

plt.legend()

plt.subplots_adjust(left=0.17)
plt.show()
```

5.3.3 score.py

```python
import math

def getElo(agent1, agent2):
    """compute expected win probability based on rating of both agents."""
    return 1/(1+math.e**((agent2.elo - agent1.elo)/400))

class Score:
    """keep track of actual the score to than later update elo with."""
    def __init__(self):
        self.score = 0

    def addWin(self):
        self.score += 1

    def addTie(self):
        self.score += 0.5

    def addLoss(self):
        pass

class PredictedScores:
    """keep track of the expected score to update elo with later."""
    def __init__(self):
        self.score = 0

    def addGame(self, this, other):
        self.score += getElo(this, other)
```

5.3.4 server.py

```
## run the elo server and handle all requests

import socket
import agent
import json
import pickle
from os.path import join as joinPath

PATH = "/media/A/MyCode/AlphaZero/elo"
print(PATH)

class Server:
    def __init__(self):
        """load data, create the server sockte and start listening"""
        self.load()
        self.serverSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.serverSock.bind(("", 2551))
        self.serverSock.listen(5)
        self.main()

    def main(self):
        """decode all incomming requests."""
        while True:
            print("waiting for connection")
            sock = self.serverSock.accept()[0]
            data = Server.getData(sock)
            if (data[0] == 1):
                #handle elo related requests
                self.update_elo(data[1], sock)
            elif (data[0] == 2):
                #handle generall data related requests.
                out = pickle.dumps(self.update_data(data[1]))

                sock.send(len(out).to_bytes(4, "little", signed=True))
                sock.send(out)

    def update_data(self, data):
        """open the correct file, find the key, specified in the request and update or change the apropriate data."""
        try:
```

```python
            with open(joinPath(PATH, "data", f"{data[0]}.json"), "r") as file:
                info = json.load(file)
        except:
            info = {}

        sub = info
        for key in data[1][:-1]:
            try:
                sub = sub[key]
            except KeyError:
                sub[key] = {}
                sub = sub[key]

        if len(data) == 3:
            sub[data[1][-1]] = data[2]

            with open(joinPath(PATH, "data", f"{data[0]}.json"), "w") as file:
                json.dump(info, file, sort_keys=True, indent=2)
            return True
        else:
            try:
                return sub[data[1][-1]]
            except KeyError:
                return None

    def getAgent(self, key):
        """get the agent asociated with a certain elo key."""
        if key == -1:
            return agent.Agent(100)

        if not key in self.agents:
            self.agents[key] = agent.Agent(100)

        return self.agents[key]

    @staticmethod
    def getData(sock):
        """read data from socket"""
        size = int.from_bytes(sock.recv(4), "little", signed=True)
        data = []
        if size == -1:
            data.append(-1)
```

```python
            data.append(int.from_bytes(sock.recv(4), "little", signed=True))
            return (1, data)

        if size == -2:
            size = int.from_bytes(sock.recv(4), "little", signed=True)
            data = pickle.loads(sock.recv(size))
            return (2, data)

        for i in range(size):
            data.append(int.from_bytes(sock.recv(4), "little", signed=True))
        return (1, data)

    def update_elo(self, data, sock):
        """change, get the elo of a certain agent or get the agent with the clossest elo. (process elo requests)"""
        deltaElo = 0

        if data[0] == -1:
            closest = None
            idx = list(self.agents.keys())[0]
            for _idx, agent in self.agents.items():
                if (_idx > 0):
                    if closest is None:
                        if data[1] < agent.elo:
                            idx = _idx
                            closest = agent

                    elif (abs(data[1] - closest.elo) > abs(data[1] - agent.elo) and data[1] < agent.elo):
                        idx = _idx
                        closest = agent
            deltaElo = idx

        else:
            agent1 = self.getAgent(data[0])
            currentElo = agent1.elo

            if len(data) == 3:
                agent2 = self.getAgent(data[1])
                agent1.addGamePrediction(agent2)
                agent.addGame(agent1, agent2, data[2])
                agent1.update(32)
```

```python

                deltaElo = abs(agent1.elo - currentElo)

            elif len(data) == 2:
                agent1.elo = data[1]
                deltaElo = abs(agent1.elo - currentElo)

            elif len(data) == 1:
                deltaElo = agent1.elo

        sock.send(int(deltaElo).to_bytes(4, "little", signed=True))
        sock.close()
        self.save()

    def load(self):
        """load elo ratings from file"""
        self.agents = {}
        try:
            with open(joinPath(PATH, "elos.json"), "r") as file:
                tmp = json.load(file)

            for key, elo in tmp.items():
                self.agents[int(key)] = agent.Agent(elo)
        except Exception as e:
            print(e)

    def save(self):
        """save elo ratings to file"""
        d = {}
        for key, agent in self.agents.items():
            d[key] = agent.elo

        with open(joinPath(PATH, "elos.json"), "w") as file:
            json.dump(d, file, sort_keys=True, indent=2)

if __name__ == "__main__":
    while True:
        try:
            server = Server()
        except: pass
```

5.4 game

5.4.1 connect4

5.4.1.1 config.hpp

```
// configure AlphaZero global variables

#pragma once
#include <log.hpp>
#include <bitset>
#include <mutex>

// check if running on unix (unix has the gpu in my runs)
#ifdef unix
#define UNIX
#endif

// set the model evaluation device (cuda or cpu)
#ifdef UNIX
#define DEVICES "cuda:0"
#endif
#ifndef UNIX
#define DEVICES "cpu"
#endif

#define OPSMode 1 // used to define what server is used
/*
+----------+-------------+
| OPSMode | Description |
+----------+-------------+
| 1        | Run Server  |
+----------+-------------+
| 2        | Run Tester  |
+----------+-------------+
*/

extern std::mutex console_mutex; // mustex used to prevent
    multiple console writes at once


#define stateSize 84             // full size of the board
    bitarray

#define U_computation(edge) (this->cpuct * edge.P * std::sqrt((
```

```
float)Nb) / (float)(1 + edge.N))
                        // comparison function for edges explained in paper


// runn setting
#define runVersion 1              // identity of the run
#define loadVersion -1              // model version to load

// Net settings
#define learningRage 0.1            // model learning rate
#define Momentum 0.9              // model momentum

// simulation setting
#define MCTSSimulations 50            // number of MCTS simulations for every action
#define cpuct_ 1.0f               // explorational constant used for U_computations
#define ProbabiliticMoves 10          // moves after start for witch to choose action probabilistically
//#define Alpha 0.9               // depricated

// memory setting
#define memory_size 30000           // amount of states in memory befor it starts training.

// self play
#define EPOCHS 1                // number of generational games to be played per thread
#define GEN_THREADS 60              // number of parallel generation threads
#define probabilitic_moves 10           // how manny moves are prabilistic in the begining of the game to aid in exploration

// training
#define Training_loops 20           // amount of batches that are generated and the model trained for per training

#define Training_batch 256            // number of states per batch
#define Training_epochs 5           // training epochs

// turney
#define Turnement_probabiliticMoves 2     // same as ProbabiliticMoves but for the model evaluation phase
```

```
#define TurneyEpochs 1               // same as EPOCHS but for the model evaluation phase
#define TurneyThreads 20             // same as GEN_THREADS but for the model evaluation phase
#define scoringThreshold 1.3          // (wins currently best player) * scoringThreshold < (new player wins)

// console
#define RenderTrainingProgress false       // whether or not the trainin progess is rendered to console
#define RenderGenAndTurneyProgress false    // whether or not the self play should be rendered to console

// Saving
#define SaverType 0                // what type of game loggs schould be created
/* +-----------+------------------------------------------------------------+
  | SaverType | Description                    |
  +-----------+------------------------------------------------------------+
  | 0         | no Saver                       |
  +-----------+------------------------------------------------------------+
  | 1         | save full state to file        |
  +-----------+------------------------------------------------------------+
  | 2         | Save taken Actions to file (int size is saved size of the int in bytes) |
  +-----------+------------------------------------------------------------+ */
#define SaverIntSize 1              // the size of an intager for the saver (1,2,3,4)


typedef std::bitset<stateSize> IDType;      // the type of the game board.
```

5.4.1.2 game.hpp

```cpp
#pragma once
/*
The game of connect for coded with game states. This allows The MCTS to simulate the game without changing it.
*/

#include <iostream>
#include <vector>
#include <list>
#include <memory>
#include <tuple>
#include <unordered_map>
#include <bitset>
#include <unordered_set>
#include <torch/torch.h>

#include "config.hpp"

// input_shape is the shape of the game board. x, y is the shape of the actual game board and z is number of stacked planes (in this case one for each
// player).
#define input_shape_x 7
#define input_shape_y 6
#define input_snape_z 2
#define action_count 42
#define action_shape 6, 7
#define boardOfset 42 // the size of a layer of the board in the buffer. (the amount of felds)(x * y)
// the actual name of the game
#define gameName "connect4"


namespace AlphaZero {
  namespace Game {
      // Game State class contains all information of a certain board possition. a board with the positions of all pieces
      // along with the curretn player. It also computes legal Actions A and win information
     class GameState {

     // the current player 1 or -1
     public: int player;
```

```
    // true if game is done (4 in a row or all filled) and false
     if not.
    public: bool done;

    // winning infromation who won and by how mutch. the tuple
   contains the following infromation in this order <current
   player win (1) or
    // current player loose (-0) or tie (0), current player
   points (1 for win -1 for loose and 0 for tie), other player
   points (- current        // player points
    public: std::tuple<int, int, int> val;

    // 84 bit bitmap that contains the current board shape. The
   first 42 bits are the positions of the player ones stones (0
   for empy, 1 for
    // stone presant) and the second 42 bits are the same for
   player -1. The 6x7 board is encoded by placing the 6 rows
   next to each other
    // starting from the fron so the top left would be bit 0 and
     the bottom right bit 41.
    public: IDType gameBoard;

    // list of allowed actions. every Action is the index of the
     bit where the stone would be placed.
    public: std::vector<int> allowedActions;


    // construct from known game state and player.
    public: GameState(IDType board, int _player);

    // construct using default state.
    public: GameState();

    // utilit function called by the constructors to avoid
   duplicate code. (all the initialization done by both of the
   constructos)
    private: void initialize(IDType board, int _player);

    // simulate an action from the current game state. (compute
   the state you would reach from this state by taking the
   following action)
    public: std::shared_ptr<GameState> takeAction(int action);

    // check weather the game is done. (will set the done
```

```
    boolean and the val tuple)
    public: void gameIsDone();

    // compute all allowed actions and writes it to the allowed
    actions list.
    protected: void getAllowedActions();

    // function that will return 1 if player 1 has a stone at
    the position specified by id, -1 if player 2 does and 0 if
    nether.
    public: int IdIndex(int id);

    //returns the id of the game state. In this case the game
    board will surfice as it contains all information. (unable to
     remove stones)
    public: IDType id();

    // renders the game state to the console in a way that is
    readable for humans.
    public: void render();
#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger
    || ModelLogger)
    // same as render() but rendes to a specified logger.
    public: void render(std::shared_ptr<spdlog::logger> logger);
#endif
    // sets the piece at id to 0, 1 or -1 (val). b is the board
    that the piece will be set to. (see gameBoard for encoding)
    public: void static IdIndex(int id, int val, IDType& b);

    // converts the game state to a tensor, that is than passed
    thought the model.
    public: torch::Tensor toTensor();

    // dose tha same thing but is more efficient for stacked
    game states during training. It will set tensor[idx] to the
    tensor representing
    // this game state.
    public: void toTensor(torch::Tensor& tensor, unsigned short
    idx=0);

    // the character representing the pieca at the position val.
     (a + if a stone can be placed there)
    private: char getPiece(int val);

    // recursive function used to determin the height of a colum
```

```
 to determin where stones can be placed. The returned int is a possible
    // placement position if the bool is true. if the bool is false the colum is full.
    private: std::pair<int, bool> getAllowedColumHeight(int);
    };

    // hash function used for hash mapes using the board as a hash key.
    struct StateHash
    {
      std::size_t operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>> const& s) const noexcept;
    };
    // returns all identidal game states and action maps (N) to the passed one.
    std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState>, std::vector<int>>> identities(std::shared_ptr<GameState> state, std::vector<int>& actionProbs);


      // The Actual game contains and handles the gamestates for the current generation Game.
    class Game {

    // pointer to the current game state.
    public: std::shared_ptr<GameState> state;


    // constructor will initialize a game state.
    public: Game();

    // reset game to initial position
    public: void reset();

    // take an action as defined by the game state
    public: void takeAction(int action);

    // human action (action is the colum that the stone should be placed in, than the action is determined)
    public: bool takeHumanAction(int action);

    // call the states render function.
    public: void render();
    };
```

```

    // Test the game.
    inline void test() {
      AlphaZero::Game::Game* game = new AlphaZero::Game::Game();

      while (!game->state->done) {
        std::vector<int> vec = { 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 };
        auto idents = identities(game->state, vec);
        idents[1].first->render();
        std::cout << "your action: ";
        int action;
        std::cin >> action;
        game->takeHumanAction(action);
#if Windows
        system("cls");
#else
        system("clear");
#endif
      }
      game->render();

      std::cout << std::endl << "the last player just won";
    }
  }
}

inline std::size_t AlphaZero::Game::StateHash::operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>>  const& s) const noexcept {
  return s.first->gameBoard.to_ullong();
}

inline int AlphaZero::Game::GameState::IdIndex(int id)
{
  if (this->gameBoard[id] == 1) {
    return 1;
  }
  else if (this->gameBoard[id + boardOfset] == 1) {
    return -1;
  }
  return 0;
}
```

```
inline void AlphaZero::Game::GameState::IdIndex(int id, int val,
    IDType& b)
{
  if (val == 0) {
    b.set(id, 0);
    b.set(id + boardOfset, 0);
    return;
  }
  if (val == -1) {
    id += boardOfset;
  }
  b.set(id, 1);
}

inline char AlphaZero::Game::GameState::getPiece(int id)
{
  std::unordered_map<int, char> renderData = { {0,'-'}, {1, 'X'
   }, {-1, 'O'} };
  if (std::find(this->allowedActions.begin(), this->
   allowedActions.end(), id) != this->allowedActions.end()) {
    return '+';
  }
  auto va = renderData[this->IdIndex(id)];
  return va;
}

inline IDType AlphaZero::Game::GameState::id()
{
  return this->gameBoard;
}

inline void AlphaZero::Game::Game::render() {
  this->state->render();
}

inline torch::Tensor AlphaZero::Game::GameState::toTensor()
{
  at::Tensor outTensor = at::zeros({ 1, input_snape_z,
   input_shape_y, input_shape_x });
  this->toTensor(outTensor);
  return outTensor;
}

inline void AlphaZero::Game::GameState::toTensor(torch::Tensor&
```

```
    tensor, unsigned short idx)
{
  unsigned short pos = 0;
  unsigned int ofset = (this->player == -1) ? 0 : boardOfset;
  for (unsigned short z = 0; z < input_snape_z; z++) {
    for (unsigned short y = 0; y < input_shape_y; y++) {
      for (unsigned short x = 0; x < input_shape_x; x++) {
        tensor[idx][z][y][x] = (float)this->gameBoard[(pos + ofset) % stateSize];
        pos++;
      }
    }
  }
}
```

5.4.1.3 game.cpp

```
#include "game.hpp"

#define columOfset 7

AlphaZero::Game::GameState::GameState(IDType board, int _player)
{
  this->initialize(board, _player);
}

AlphaZero::Game::GameState::GameState()
{
  this->initialize(IDType(), 1);
}

void AlphaZero::Game::GameState::initialize(IDType board, int
    _player)
{
  this->gameBoard = board;
  this->player = _player;
  this->getAllowedActions();
  this->gameIsDone();
}

std::shared_ptr<AlphaZero::Game::GameState> AlphaZero::Game::
    GameState::takeAction(int action)
{
  IDType newBoard = this->gameBoard;
  GameState::IdIndex(action, this->player, newBoard);

  std::shared_ptr<GameState> newState = std::make_shared<
    GameState>(newBoard, -this->player);
  return newState;
}

void AlphaZero::Game::GameState::gameIsDone()
{
  std::vector<std::vector<int>> winOptions = {
    /*
    +----+----+----+----+----+----+----+
    | 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 |
+----+----+----+----+----+----+----+
*/
//horizontal
{0,1,2,3},
{1,2,3,4},
{2,3,4,5},
{3,4,5,6},

{7,8,9,10},
{8,9,10,11},
{9,10,11,12},
{10,11,12,13},

{14,15,16,17},
{15,16,17,18},
{16,17,18,19},
{17,18,19,20},

{21,22,23,24},
{22,23,24,25},
{23,24,25,26},
{24,25,26,27},

{28,29,30,31},
{29,30,31,32},
{30,31,32,33},
{31,32,33,34},

{35,36,37,38},
{36,37,38,39},
{37,38,39,40},
{38,39,40,41},
//vertical
{0 , 7,14,21},
{7 ,14,21,28},
{14,21,28,35},

{1 ,8 ,15,22},
```

```
    {8 ,15,22,29},
    {15,22,29,36},

    {2,9,16,23},
    {9,16,23,30},
    {16,23,30,37},

    {3 ,10,17,24},
    {10,17,24,31},
    {17,24,31,38},

    {4 ,11,18,25},
    {11,18,25,32},
    {18,25,32,39},

    {5 ,12,19,26},
    {12,19,26,33},
    {19,26,33,40},

    {6 ,13,20,27},
    {13,20,27,34},
    {20,27,34,41},

    //diagonal topleft-bottomRight
    {14,22,30,38},

    {7, 15,23,31},
    {15,23,31,39},

    {0, 8, 16,24},
    {8, 16,24,32},
    {16,24,32,40},

    {1, 9 ,17,25},
    {9, 17,25,33},
    {17,25,33,41},

    {2, 10,18,26},
    {10,18,26,34},

    {3, 11,19,27},

    //diagonal topright-bottomleft
    {3, 9, 15,21},

```

```
    {4, 10,16,22},
    {10,16,22,28},

    {5, 11,17,23},
    {11,17,23,29},
    {17,23,29,35},

    {6, 12,18,24},
    {12,18,24,30},
    {18,24,30,36},

    {13,19,25,31},
    {19,25,31,37},

    {20,26,32,38},
  };
  bool tie = true;
  for (int idx = 0; idx < action_count; idx++) {
    if (this->IdIndex(idx) == 0) {
      tie = false;
      break;
    }
  }
  if (tie) {
    this->done = true;
    this->val = { 0,0,0 };
    return;
  }
  for (auto option : winOptions) {
    int count = 0;
    for (int pos : option) {
      count += this->IdIndex(pos);
    }
    if (count == -4 * this->player) {
      this->done = true;
      this->val = { -1, -1, 1 }; // winForThisPlayer, points for this player, points for other player
      return;
    }
  }
  this->done = false;
  this->val = { 0, 0, 0 };
}

inline std::pair<int, bool> AlphaZero::Game::GameState::
```

```
    getAllowedColumHeight(int idx) {
  if (this->IdIndex(idx) != 0) {
    return { idx, false };
  }
  if (idx >= 35) {
    return {idx, true};
  }
  else if (this->IdIndex(idx + columOfset)!=0) {
    return { idx, true };
  }
  else {
    return this->getAllowedColumHeight(idx + columOfset);
  }
}

void AlphaZero::Game::GameState::getAllowedActions()
{
  this->allowedActions.clear();
  for (int idx = 0; idx < 7; idx++) {
    std::pair<int, bool> data = this->getAllowedColumHeight(idx)
    ;
    if (data.second) {
      this->allowedActions.push_back(data.first);
    }
  }
}

void AlphaZero::Game::GameState::render()
{
  console_mutex.lock();
  for (int row = 0; row < action_count;) {
    for (int iter = 0; iter < 7; iter++) {
      std::cout << this->getPiece(row) << " ";
      row++;
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
  console_mutex.unlock();
}

#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger
    || ModelLogger)
void AlphaZero::Game::GameState::render(std::shared_ptr<spdlog::
    logger> logger)
```

```
{
  for (int idx = 0; idx < 6; idx++) {
    char line1[13] = {
      this->getPiece(0 + columOfset * idx), ' ',
      this->getPiece(1 + columOfset * idx), ' ',
      this->getPiece(2 + columOfset * idx), ' ',
      this->getPiece(3 + columOfset * idx), ' ',
      this->getPiece(4 + columOfset * idx), ' ',
      this->getPiece(5 + columOfset * idx), ' ',
      this->getPiece(6 + columOfset * idx)
    };
    logger->info(line1);
  }
}
#endif

AlphaZero::Game::Game::Game()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::reset()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::takeAction(int action)
{
  auto newState = this->state->takeAction(action);
  this->state = newState;
}

bool AlphaZero::Game::Game::takeHumanAction(int action)
{
  for (auto const& allowed : this->state->allowedActions) {
    if ((allowed - action) % 7 == 0) {
      this->takeAction(allowed);
      return true;
    }
  }
  return false;
}

inline std::pair<std::shared_ptr<AlphaZero::Game::GameState>,
    std::vector<int>> mirrorGameState(std::shared_ptr<AlphaZero::
```

```cpp
    Game::GameState> state, std::vector<int>& actionProbs) {
  IDType boardBuffer;

  std::vector<int> probs = {
     actionProbs[6],  actionProbs[5],  actionProbs[4],  actionProbs[3],  actionProbs[2],  actionProbs[1],  actionProbs[0],
     actionProbs[13], actionProbs[12], actionProbs[11], actionProbs[10], actionProbs[9],  actionProbs[8],  actionProbs[7],
     actionProbs[20], actionProbs[19], actionProbs[18], actionProbs[17], actionProbs[16], actionProbs[15], actionProbs[14],
     actionProbs[27], actionProbs[26], actionProbs[25], actionProbs[24], actionProbs[23], actionProbs[22], actionProbs[21],
     actionProbs[34], actionProbs[33], actionProbs[32], actionProbs[31], actionProbs[30], actionProbs[29], actionProbs[28],
     actionProbs[41], actionProbs[40], actionProbs[39], actionProbs[38], actionProbs[37], actionProbs[36], actionProbs[35]
  };
#define assignStateSinge(idx1, idx2)AlphaZero::Game::GameState::IdIndex(idx1, state->IdIndex(idx2), boardBuffer)
#define assignState(idx1, idx2)assignStateSinge(idx1, idx2); assignStateSinge(idx2, idx1);


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

  return { std::make_shared<AlphaZero::Game::GameState>(boardBuffer, state->player), probs };
```

```
}

std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState
    >, std::vector<int>>> AlphaZero::Game::identities(std::
    shared_ptr<GameState> state, std::vector<int>& probs)
{
  std::vector<std::pair<std::shared_ptr<AlphaZero::Game::
    GameState>, std::vector<int>>> idents(2);
  int id = 0;
  idents[0] = { state, probs };
  idents[1] = mirrorGameState(state, probs);
  return idents;
}
```

5.4.2 othello

5.4.2.1 config.hpp

```
#pragma once
#include <log.hpp>
#include <bitset>
#include <mutex>

#ifdef unix
#define UNIX
#endif

#ifdef UNIX
#define DEVICES "cuda:0"
#endif
#ifndef UNIX
#define DEVICES "cpu"
#endif

#define OPSMode 1

extern std::mutex console_mutex;
extern std::mutex rand_mutex;

/*
+----------+-------------+
| OPSMode | Description |
+----------+-------------+
| 1        | Run Server  |
+----------+-------------+
| 2        | Run Tester  |
+----------+-------------+
*/

#define stateSize 128
#define Training true

#define U_computation(edge) (this->cpuct * edge.P * std::sqrt((
    float)Nb) / (float)(1 + edge.N))


// runn setting
#define runVersion 1
#define loadVersion -1
```

```

// Net settings
#define reg_const 0.0001
#define learningRage 0.1
#define Momentum 0.9

// simulation setting
#define MCTSSimulations 50
#define cpuct_ 1.0f
#define ProbabiliticMoves 10
#define Alpha 0.9
#define EPSILON 0.2f

// memory setting
#define memory_size 30000

// self play
#define EPOCHS 1
#define GEN_THREADS 60
#define probabilitic_moves 10 // how manny moves are prabilistic
    in the begining of the game to aid in exploration

// training
#define Training_loops 20
#define Training_batch 256
#define Training_epochs 5

// turney
#define Turnement_probabiliticMoves 2
#define TurneyEpochs 1
#define TurneyThreads 20
#define scoringThreshold 1.3

// console
#define RenderTrainingProgress false
#define RenderGenAndTurneyProgress false
//#define RenderGameProgress true;

// Saving
#define SaverType 0
/*
   +-----------+-------------------------------------------------------------------------------

  | SaverType | Description
                            |
```

```
+------------+------------------------------------------------------------------
| 0          | no Saver                                  |
+------------+------------------------------------------------------------------
| 1          | save full state to file                   |
+------------+------------------------------------------------------------------
| 2          | Save taken Actions to file (int size is saved size of the int in bytes) |
+------------+------------------------------------------------------------------
*/
#define SaverIntSize 1


typedef std::bitset<stateSize> IDType;
```

5.4.2.2 game.hpp

```
#pragma once
/*
this is the alpha Zero game for Orthello
*/

#include <iostream>
#include <vector>
#include <list>
#include <memory>
#include <tuple>
#include <unordered_map>
#include <bitset>
#include <unordered_set>
#include <torch/torch.h>

#include "config.hpp"


#define input_shape_x 8
#define input_shape_y 8
#define input_snape_z 2
#define action_count 64
#define action_shape 8, 8
#define boardOfset 64 // the size of a layer of the board in the
    buffer. (the amount of felds)
#define gameName "Orthello"


namespace AlphaZero {
  namespace Game {
    class GameState {
    public: int player;
    public: bool done;
    public: std::tuple<int, int, int> val;
    public: IDType gameBoard;
    public: std::vector<int> allowedActions;

    public: GameState(IDType board, int _player);
    public: GameState();
    private: void initialize(IDType board, int _player);
    public: std::shared_ptr<GameState> takeAction(int action);
    public: void gameIsDone();
    protected: void getAllowedActions();
```

```
    public: int IdIndex(int id);
    public: IDType id();
    public: void render();
#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger)
    public: void render(std::shared_ptr<spdlog::logger> logger);
#endif
    public: void static IdIndex(int id, int val, IDType& b);
    public: torch::Tensor toTensor();
    public: void toTensor(torch::Tensor& tensor, unsigned short idx=0);
    public: char getPiece(int val);
    private: std::vector<int> getFlipActions(int x, int y, int dx, int dy);
    private: bool hasAjacentStones(int const& x, int const& y);
    };
    struct StateHash
    {
      std::size_t operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>> const& s) const noexcept;
    };
    // optimization function its not a problem if not all are found
    std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState>, std::vector<int>>> identities(std::shared_ptr<GameState> state, std::vector<int>& actionProbs);

    class Game {
    public: std::shared_ptr<GameState> state;

    public: Game();
    public: void reset();
    public: void takeAction(int action);
    public: bool takeHumanAction(int action);
    public: void render();
    };

    inline void test() {
      AlphaZero::Game::Game game;

      while (!game.state->done) {
        auto idx = std::rand() % game.state->allowedActions.size();
        auto action = game.state->allowedActions[idx];
        game.render();
```

```
        game.takeAction(action);
        char arr[100];

        //system("cls");
      }
      game.render();

      std::cout << std::endl << "the last player just won";
    }
    std::pair<int, int> to2dPos(int pos);
    int from2dPos(int const& x, int const& y);

    void renderStates(std::vector<GameState*> states);
  }
}

inline std::size_t AlphaZero::Game::StateHash::operator()(std::pair<std::shared_ptr<GameState>, std::vector<int>> const& s) const noexcept {
  return s.first->gameBoard.to_ullong();
}

inline int AlphaZero::Game::GameState::IdIndex(int id)
{
  if (this->gameBoard[id] == 1) {
    return 1;
  }
  else if (this->gameBoard[id + boardOfset] == 1) {
    return -1;
  }
  return 0;
}

inline void AlphaZero::Game::GameState::IdIndex(int id, int val, IDType& b)
{
  int otherId;
  if (val == 0) {
    b.set(id, 0);
    b.set(id + boardOfset, 0);
    return;
  }
  if (val == 1)
  {
    otherId = id + boardOfset;
```

```
  }
  if (val ==-1) {
    otherId = id;
    id += boardOfset;
  }
  b.set(id, 1);
  b.set(otherId, 0);
}

inline char AlphaZero::Game::GameState::getPiece(int id)
{
  std::unordered_map<int, char> renderData = { {0,'-'}, {1, 'X'}, {-1, 'O'} };
  if (std::find(this->allowedActions.begin(), this->allowedActions.end(), id) != this->allowedActions.end()) {
    return '+';
  }
  auto va = renderData[this->IdIndex(id)];
  return va;
}

inline IDType AlphaZero::Game::GameState::id()
{
  return this->gameBoard;
}

inline void AlphaZero::Game::Game::render() {
  this->state->render();
}

inline torch::Tensor AlphaZero::Game::GameState::toTensor()
{
  at::Tensor outTensor = at::zeros({ 1, input_snape_z, input_shape_y, input_shape_x });
  this->toTensor(outTensor);
  return outTensor;
}

inline void AlphaZero::Game::GameState::toTensor(torch::Tensor& tensor, unsigned short idx)
{
  unsigned short pos = 0;
  unsigned int ofset = (this->player == -1) ? 0 : boardOfset;
  for (unsigned short z = 0; z < input_snape_z; z++) {
    for (unsigned short y = 0; y < input_shape_y; y++) {
```

```
      for (unsigned short x = 0; x < input_shape_x; x++) {
        tensor[idx][z][y][x] = (float)this->gameBoard[(pos + ofset) % stateSize];
        pos++;
      }
    }
  }
}

inline std::pair<int, int> AlphaZero::Game::to2dPos(int pos)
{
  int x = pos % input_shape_x;
  int y = (pos - x) / input_shape_x;
  return { x,y };
}

inline int AlphaZero::Game::from2dPos(int const& x, int const& y)
{
  return x + y * input_shape_x;
}
```

5.4.2.3 game.cpp

```cpp
#include "game.hpp"

#define columOfset 8

std::vector<std::pair<int, int>> ajacentDirections = { {1,0},{-1,0},{0,1},{0,-1} };
std::vector<std::pair<int, int>> flipDirections = { {1,0},{-1,0},{0,1}, {0,-1}, {1,1},{1,-1},{-1,-1},{-1,1} };

bool isValidPostition(int const& x, int const& y)
{
  if (x < 0) { return false; }
  if (y < 0) { return false; }
  if (x >= input_shape_x) { return false; }
  if (y >= input_shape_y) { return false; }
  return true;
}

std::vector<int> AlphaZero::Game::GameState::getFlipActions(int x, int y, int dx, int dy)
{
  std::vector<int> tmp;
  std::pair<int, int> a = { x + dx, y + dy };
  auto value = this->player;
  while (isValidPostition(a.first, a.second))
  {
    auto idx = from2dPos(a.first, a.second);
    auto otherValue = this->IdIndex(idx);
    if (value == -otherValue)
    {
      tmp.push_back(idx);
    }
    else if (value == otherValue)
    {
      return tmp;
    }
    else
    {
      return std::vector<int>();
    }
    a = { a.first + dx, a.second + dy };
  }
  return std::vector<int>();
```

```
}

bool AlphaZero::Game::GameState::hasAjacentStones(int const& x,
    int const& y)
{
  int otherX, otherY;
  for (auto const& direction : ajacentDirections)
  {
    otherX = x + direction.first;
    otherY = y + direction.second;
    if (isValidPostition(otherX, otherY))
    {
      if (this->IdIndex(from2dPos(otherX, otherY)) != 0)
      {
        return true;
      }
    }
  }
  return false;
}

AlphaZero::Game::GameState::GameState(IDType board, int _player)
{
  this->initialize(board, _player);
}

AlphaZero::Game::GameState::GameState()
{
  IDType board;
  this->IdIndex(27, -1, board);
  this->IdIndex(28,  1, board);
  this->IdIndex(35,  1, board);
  this->IdIndex(36, -1, board);
  this->initialize(board, 1);
}

void AlphaZero::Game::GameState::initialize(IDType board, int
    _player)
{
  this->gameBoard = board;
  this->player = _player;
  this->getAllowedActions();
  this->gameIsDone();
}

```

```
std::shared_ptr<AlphaZero::Game::GameState> AlphaZero::Game::GameState::takeAction(int action)
{
  IDType newBoard = this->gameBoard;
  GameState::IdIndex(action, this->player, newBoard);
  std::pair<int, int> pos = to2dPos(action);
  for (auto const& direction : flipDirections)
  {
    auto toFlip = getFlipActions(pos.first, pos.second, direction.first, direction.second);
    for (auto const& pos : toFlip)
    {
      GameState::IdIndex(pos, this->player, newBoard);
    }
  }

  std::shared_ptr<GameState> newState = std::make_shared<GameState>(newBoard, -this->player);

  if (newState->allowedActions.size())
    return newState;

  std::shared_ptr<GameState> newerState = std::make_shared<GameState>(newBoard, this->player);
  return newerState;
}

void AlphaZero::Game::GameState::gameIsDone()
{
  int thisPlayerPoints = 0, otherPlayerPoints = 0;
  this->done = true;
  for (int x = 0; x < input_shape_x; x++)
  {
    for (int y = 0; y < input_shape_y; y++)
    {
      auto val = this->IdIndex(x + y * columOfset);
      if (val == 0 && this->allowedActions.size())
      {
        this->done = false;
      }
      else if (val == this->player)
      {
        thisPlayerPoints++;
      }
      else if (val == -this->player)
```

```
        {
          otherPlayerPoints++;
        }
      }
    }
    if (this->done)
    {
      int win = 0;
      if (thisPlayerPoints > otherPlayerPoints)
      {
        win = 1;
      }
      else if (thisPlayerPoints < otherPlayerPoints)
      {
        win = -1;
      }
      this->val = { win, thisPlayerPoints, otherPlayerPoints };
    }
    else
    {
      this->val = { 0, 0, 0 };
    }
}

void AlphaZero::Game::GameState::getAllowedActions()
{
    this->allowedActions.clear();
    for (int x = 0; x < input_shape_x; x++)
    {
      for (int y = 0; y < input_shape_y; y++)
      {
        if (this->IdIndex(from2dPos(x, y)) == 0 &&
      hasAjacentStones(x, y))
        {
          for (auto const& direction : flipDirections)
          {
            auto toFlip = getFlipActions(x, y, direction.first,
      direction.second);
            if (toFlip.size())
            {
              this->allowedActions.push_back(from2dPos(x, y));
              break;
            }
          }
        }
```

```cpp
    }
  }
}

void AlphaZero::Game::GameState::render()
{
  std::vector<GameState*> state = { this };
  renderStates(state);
}

#if (MainLogger || MCTSLogger || MemoryLogger || ProfileLogger || ModelLogger)
void AlphaZero::Game::GameState::render(std::shared_ptr<spdlog::logger> logger)
{
  for (int idx = 0; idx < 6; idx++) {
    char line1[13] = {
      this->getPiece(0 + columOfset * idx), ' ',
      this->getPiece(1 + columOfset * idx), ' ',
      this->getPiece(2 + columOfset * idx), ' ',
      this->getPiece(3 + columOfset * idx), ' ',
      this->getPiece(4 + columOfset * idx), ' ',
      this->getPiece(5 + columOfset * idx), ' ',
      this->getPiece(6 + columOfset * idx)
    };
    logger->info(line1);
  }
}
#endif

AlphaZero::Game::Game::Game()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::reset()
{
  this->state = std::make_shared<GameState>();
}

void AlphaZero::Game::Game::takeAction(int action)
{
  auto newState = this->state->takeAction(action);
  this->state = newState;
}
```

```cpp

bool AlphaZero::Game::Game::takeHumanAction(int action)
{
  this->takeAction(action);
  return true;
}

#define assign(idx1, idx2)AlphaZero::Game::GameState::IdIndex(
    idx1, state->IdIndex(idx2), boardBuffer)
#define assignState(idx1, idx2)assign(idx1, idx2);assign(idx2,
    idx1)

inline std::pair<std::shared_ptr<AlphaZero::Game::GameState>,
    std::vector<int>> mirrorGameState(std::shared_ptr<AlphaZero::
    Game::GameState> state, std::vector<int> const& actionProbs)
    {
  IDType boardBuffer;

  std::vector<int> probs = {
     actionProbs[7],  actionProbs[6],  actionProbs[5],
    actionProbs[4],  actionProbs[3],  actionProbs[2],
    actionProbs[1],  actionProbs[0],
     actionProbs[15], actionProbs[14], actionProbs[13],
    actionProbs[12], actionProbs[11], actionProbs[10],
    actionProbs[9],  actionProbs[8],
     actionProbs[23], actionProbs[22], actionProbs[21],
    actionProbs[20], actionProbs[19], actionProbs[18],
    actionProbs[17], actionProbs[16],
     actionProbs[31], actionProbs[30], actionProbs[29],
    actionProbs[28], actionProbs[27], actionProbs[26],
    actionProbs[25], actionProbs[24],
     actionProbs[39], actionProbs[38], actionProbs[37],
    actionProbs[36], actionProbs[35], actionProbs[34],
    actionProbs[33], actionProbs[32],
     actionProbs[47], actionProbs[46], actionProbs[45],
    actionProbs[44], actionProbs[43], actionProbs[42],
    actionProbs[41], actionProbs[40],
     actionProbs[55], actionProbs[54], actionProbs[53],
    actionProbs[52], actionProbs[51], actionProbs[50],
    actionProbs[49], actionProbs[48],
     actionProbs[63], actionProbs[62], actionProbs[61],
    actionProbs[60], actionProbs[59], actionProbs[58],
    actionProbs[57], actionProbs[56]
  };

```

```
  assignState(0, 7);   assignState(1, 6);   assignState(2, 5);
    assignState(3, 4);
  assignState(8, 15);  assignState(9, 14);  assignState(10, 13);
    assignState(11, 12);
  assignState(16, 23); assignState(17, 22); assignState(18, 21);
    assignState(19, 20);
  assignState(24, 31); assignState(25, 30); assignState(26, 29);
    assignState(27, 28);
  assignState(32, 39); assignState(33, 38); assignState(34, 37);
    assignState(35, 36);
  assignState(40, 47); assignState(41, 46); assignState(42, 45);
    assignState(43, 44);
  assignState(48, 55); assignState(49, 54); assignState(50, 53);
    assignState(51, 52);
  assignState(56, 63); assignState(57, 62); assignState(58, 61);
    assignState(59, 60);

  return { std::make_shared<AlphaZero::Game::GameState>(
   boardBuffer, state->player), probs };
}

inline std::pair<std::shared_ptr<AlphaZero::Game::GameState>,
   std::vector<int>> rotateGameState(std::shared_ptr<AlphaZero::
   Game::GameState> state, std::vector<int> const& actionProbs)
{
  IDType boardBuffer;

  std::vector<int> probs = {
    actionProbs[56], actionProbs[48], actionProbs[40],
   actionProbs[32], actionProbs[24], actionProbs[16],
   actionProbs[8],  actionProbs[0],
    actionProbs[57], actionProbs[49], actionProbs[41],
   actionProbs[33], actionProbs[25], actionProbs[17],
   actionProbs[9],  actionProbs[1],
    actionProbs[58], actionProbs[50], actionProbs[42],
   actionProbs[34], actionProbs[26], actionProbs[18],
   actionProbs[10], actionProbs[2],
    actionProbs[59], actionProbs[51], actionProbs[43],
   actionProbs[35], actionProbs[27], actionProbs[19],
   actionProbs[11], actionProbs[3],
    actionProbs[60], actionProbs[52], actionProbs[44],
   actionProbs[36], actionProbs[28], actionProbs[20],
   actionProbs[12], actionProbs[4],
    actionProbs[61], actionProbs[53], actionProbs[45],
```

```
    actionProbs[37], actionProbs[29], actionProbs[21], actionProbs[13], actionProbs[5],
    actionProbs[62], actionProbs[54], actionProbs[46], actionProbs[38], actionProbs[30], actionProbs[22], actionProbs[14], actionProbs[6],
    actionProbs[63], actionProbs[55], actionProbs[47], actionProbs[39], actionProbs[31], actionProbs[23], actionProbs[15], actionProbs[7]
  };

  auto a = state->IdIndex(0);
  assign(0, 56);  assign(1, 48);  assign(2, 40);  assign(3, 32);  assign(4, 24);  assign(5, 16);  assign(6, 8);   assign(7, 0);
  assign(8, 57);  assign(9, 49);  assign(10, 41); assign(11, 33); assign(12, 25); assign(13, 17); assign(14, 9);  assign(15, 1);
  assign(16, 58); assign(17, 50); assign(18, 42); assign(19, 34); assign(20, 26); assign(21, 18); assign(22, 10); assign(23, 2);
  assign(24, 59); assign(25, 51); assign(26, 43); assign(27, 35); assign(28, 27); assign(29, 19); assign(30, 11); assign(31, 3);
  assign(32, 60); assign(33, 52); assign(34, 44); assign(35, 36); assign(36, 28); assign(37, 20); assign(38, 12); assign(39, 4);
  assign(40, 61); assign(41, 53); assign(42, 45); assign(43, 37); assign(44, 29); assign(45, 21); assign(46, 13); assign(47, 5);
  assign(48, 62); assign(49, 54); assign(50, 46); assign(51, 38); assign(52, 30); assign(53, 22); assign(54, 14); assign(55, 6);
  assign(56, 63); assign(57, 55); assign(58, 47); assign(59, 39); assign(60, 31); assign(61, 23); assign(62, 15); assign(63, 7);

  return { std::make_shared<AlphaZero::Game::GameState>(boardBuffer, state->player), probs };
}

#undef assignState
#undef assign

bool canBeAddedToIdentities(std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState>, std::vector<int>>>
```

```
    const& idents, std::pair<std::shared_ptr<AlphaZero::Game::
    GameState>, std::vector<int>>const& data)
{
  auto pos = std::find(idents.begin(), idents.end(), data);
  return pos == idents.end();
}

std::vector<std::pair<std::shared_ptr<AlphaZero::Game::GameState
    >, std::vector<int>>> AlphaZero::Game::identities(std::
    shared_ptr<GameState> state, std::vector<int>& probs)
{
  std::vector<std::pair<std::shared_ptr<AlphaZero::Game::
    GameState>, std::vector<int>>> idents;
  int idx = 0;
  std::pair<std::shared_ptr<GameState>, std::vector<int>> data =
     { state, probs };
  std::pair<std::shared_ptr<GameState>, std::vector<int>>
    mirrored = mirrorGameState(state, probs);
  for (unsigned short iter = 0; iter < 3; iter++)
  {
    if (canBeAddedToIdentities(idents, data))
      idents.push_back(data);
    if (canBeAddedToIdentities(idents, mirrored))
      idents.push_back(mirrored);

    data = rotateGameState(data.first, data.second);
    mirrored = rotateGameState(mirrored.first, mirrored.second);
  }
  if (canBeAddedToIdentities(idents, data))
    idents.push_back(data);
  if (canBeAddedToIdentities(idents, mirrored))
    idents.push_back(mirrored);
  return idents;
}

void AlphaZero::Game::renderStates(std::vector<GameState*>
    states)
{
  console_mutex.lock();
  for (int y = 0; y < input_shape_y; y++)
  {
    for (auto const& state : states)
    {
      for (int x = 0; x < input_shape_x; x++)
      {
```

```
        std::cout << state->getPiece(from2dPos(x, y)) << " ";
      }
      std::cout << "\t\t";
    }
    std::cout << std::endl;
  }
  std::cout << std::endl;
  console_mutex.unlock();
}
```

5.5 CMakeLists.txt

```
# CMakeList.txt : Top-level CMake project file, do global
  configuration
# and include sub-projects here.
#
cmake_minimum_required (VERSION 3.8)

project ("AlphaZeroPytorch")
find_package(CUDA 7.0)
if (NOT CUDA_VERSION_STRING EQUAL "")
  message(STATUS "did not find cuda")
else()
  message(STATUS "Found CUDA ${CUDA_VERSION_STRING} at ${
   CUDA_TOOLKIT_ROOT_DIR}")
endif()

# Include sub-projects.
#add_subdirectory ("AlphaZeroPytorch")
```

5.6 README.md

```
# AlphaZero
Server baised alpha Zero server and client.
```

5.7 serch.sh

```
clear
echo grep -r $1 .
```

6 Demos

6.1 Matura-AlphaZero-demos

6.1.1 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0.0)
project(AlphaZeroDemos VERSION 0.1.0)
set(CXX_STANDARD 14)

include(CTest)
enable_testing()
message(STATUS ${CMAKE_MODULE_PATH})

set(libtorch_hash "5b5ea3067f878dea051f6fcd8fb00338f55517cb6baecdc810983a814e030845")

# Download libtorch from offical website

if (EXISTS "${PROJECT_SOURCE_DIR}/dependencies/libtorch.zip" OR EXISTS "${PROJECT_SOURCE_DIR}/dependencies/libtorch")
  message(STATUS "libtorch already downloaded")
else ()
  message(STATUS "downloading libtorch")
  file(
    DOWNLOAD
    "https://download.pytorch.org/libtorch/cpu/libtorch-cxx11-abi-shared-with-deps-1.11.0%2Bcpu.zip"
    "${PROJECT_SOURCE_DIR}/dependencies/libtorch.zip"
  )
  file(SHA256 "${PROJECT_SOURCE_DIR}/dependencies/libtorch.zip" libtorch_checksum)
  if (libtorch_checksum MATCHES "${libtorch_hash}")
    message(STATUS "libtorch checksum is valid")
  else()
    message(FATAL_ERROR "libtorch checksum is not valid")
  endif()
endif()
 #unzip libtorch
if (EXISTS "${PROJECT_SOURCE_DIR}/dependencies/libtorch")
  message(STATUS "libtorch already installed")
else()
  message(STATUS "installing libtorch")
```

```
  file(ARCHIVE_EXTRACT INPUT "${PROJECT_SOURCE_DIR}/dependencies/libtorch.zip" DESTINATION "${PROJECT_SOURCE_DIR}/dependencies")
  file(REMOVE "${PROJECT_SOURCE_DIR}/dependencies/libtorch.zip")
  message(STATUS "libtorch installed")
endif()
message(STATUS "libtorch Path:\t" "${PROJECT_SOURCE_DIR}/dependencies/libtorch")
set (Torch_DIR "${PROJECT_SOURCE_DIR}/dependencies/libtorch/share/cmake/Torch")


find_package(Torch REQUIRED Torch_DIR)
set(DCMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${TORCH_CXX_FLAGS}")

find_package(colormap REQUIRED INTERFACE)

find_package(wxWidgets REQUIRED gl core base OPTIONAL_COMPONENTS net)
include(${wxWidgets_USE_FILE})

file(GLOB SharedFiles
  "include/*"
  "scr/*"
)
file(GLOB UnsupervisedFiles
  "unsupervised/src/*"
  "unsupervised/include/*"
)
file(GLOB SupervisedFiles
  "supervised/src/*"
  "supervised/include/*"
)

add_executable(VectorQuantization main.cpp ${SharedFiles} ${UnsupervisedFiles})
target_include_directories(VectorQuantization PUBLIC "include/" "unsupervised/include")
target_link_libraries(VectorQuantization PRIVATE ${wxWidgets_LIBRARIES})

add_executable(Supervised supervised.cpp ${SharedFiles} ${SupervisedFiles})
target_include_directories(Supervised PUBLIC "include/" "supervised/include")
```

```
target_link_libraries(Supervised PRIVATE ${wxWidgets_LIBRARIES}
    ${TORCH_LIBRARIES})
target_link_libraries(Supervised INTERFACE ${colormap_DIR})
message(STATUS "colormap at: " ${colormap_DIR})
#set_property(TARGET Supervised PROPERTY CXX_STANDARD 23)

set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})
include(CPack)
```

6.1.2 main.cpp

```cpp
#include <wx/sizer.h>
#include <wx/timer.h>
#include <config.hpp>
#include <cluster.hpp>

class BasicDrawPane;

class RenderTimer : public wxTimer
{
    BasicDrawPane* pane;
public:
    RenderTimer(BasicDrawPane* pane);
    void Notify();
    void start();
};


class BasicDrawPane : public wxPanel
{
    VQ::Cluster* cluster;
public:
    BasicDrawPane(wxFrame* parent);
    ~BasicDrawPane();

    void paintEvent(wxPaintEvent& evt);
    void paintNow();
    void render( wxDC& dc );

    DECLARE_EVENT_TABLE()
};

class MyFrame;

class MyApp: public wxApp
{
    bool OnInit();

    MyFrame* frame;
public:

};


```

```
RenderTimer::RenderTimer(BasicDrawPane* pane) : wxTimer()
{
    RenderTimer::pane = pane;
}

void RenderTimer::Notify()
{
    pane->Refresh();
}

void RenderTimer::start()
{
    wxTimer::Start(10);
}

IMPLEMENT_APP(MyApp)

class MyFrame : public wxFrame
{
    RenderTimer* timer;
    BasicDrawPane* drawPane;

public:
    MyFrame() : wxFrame((wxFrame *)NULL, -1,  wxT("Hello wxDC"),
     wxPoint(50,50), wxSize(WIDTH,HEIGHT))
    {
        wxBoxSizer* sizer = new wxBoxSizer(wxHORIZONTAL);
        drawPane = new BasicDrawPane( this );
        sizer->Add(drawPane, 1, wxEXPAND);
        SetSizer(sizer);

        timer = new RenderTimer(drawPane);
        Show();
        timer->start();
    }
    ~MyFrame()
    {
        delete timer;
    }
    void onClose(wxCloseEvent& evt)
    {
        timer->Stop();
        evt.Skip();
    }
    DECLARE_EVENT_TABLE()
```

```
};


BEGIN_EVENT_TABLE(MyFrame, wxFrame)
EVT_CLOSE(MyFrame::onClose)
END_EVENT_TABLE()

bool MyApp::OnInit()
{
    frame = new MyFrame();
    frame->Show();

    return true;
}


BEGIN_EVENT_TABLE(BasicDrawPane, wxPanel)
EVT_PAINT(BasicDrawPane::paintEvent)
END_EVENT_TABLE()



BasicDrawPane::BasicDrawPane(wxFrame* parent) :
wxPanel(parent)
{
  auto a = wxGREEN_BRUSH;
  std::vector<const wxBrush*> brushes;
  brushes.push_back(wxGREEN_BRUSH);
  brushes.push_back(wxRED_BRUSH);
  brushes.push_back(wxBLUE_BRUSH);
  this->cluster = new VQ::Cluster(POINT_COUNT, (double)WIDTH, (double)HEIGHT, brushes);
}
BasicDrawPane::~BasicDrawPane(){
  delete this->cluster;
}


void BasicDrawPane::paintEvent(wxPaintEvent& evt)
{
    wxPaintDC dc(this);
    render(dc);
}

void BasicDrawPane::paintNow()
```

```
{
    wxClientDC dc(this);
    render(dc);
}

void BasicDrawPane::render( wxDC& dc )
{
    this->cluster->update();

    dc.SetBackground( *wxWHITE_BRUSH );
    dc.SetBrush( *wxBLUE_BRUSH);
    dc.Clear();
    this->cluster->render(dc);
}
```

6.1.3 README.md

````
Installation
------------


prerequisits
  - [wxWidgets](https://wiki.wxwidgets.org/Getting_Started_with_wxWidgets)
  - [colormap](https://github.com/JulianWww/colormap)


to install run:
```
$ git clone https://github.com/JulianWww/Matura-AlphaZero-demos
$ cd Matura-AlphaZero-demos && mkdir build && cd build
$ cmake .. && make
```
````

6.1.4 supervised.cpp

```
#include <model.hpp>
#include <wx/wx.h>
#include <wx/sizer.h>
#include <wx/timer.h>
#include <config.hpp>

class BasicDrawPane;

class RenderTimer : public wxTimer
{
    BasicDrawPane* pane;
public:
    RenderTimer(BasicDrawPane* pane);
    void Notify();
    void start();
};

class MyFrame;
class BasicDrawPane : public wxPanel
{
  private: MyFrame* parent;
  private: SL::Model* model;
public:
    BasicDrawPane(MyFrame* parent);
    ~BasicDrawPane();

    void paintEvent(wxPaintEvent& evt);
    void paintNow();
    void render( wxDC& dc );

    DECLARE_EVENT_TABLE()
};

class MyApp: public wxApp
{
    bool OnInit();

    MyFrame* frame;
public:

};


```

```
RenderTimer::RenderTimer(BasicDrawPane* pane) : wxTimer()
{
    RenderTimer::pane = pane;
}

void RenderTimer::Notify()
{
    pane->Refresh();
}

void RenderTimer::start()
{
    wxTimer::Start(10);
}

IMPLEMENT_APP(MyApp)

class MyFrame : public wxFrame
{
    RenderTimer* timer;
    BasicDrawPane* drawPane;

public:
    wxBoxSizer* sizer;
    MyFrame() : wxFrame((wxFrame *)NULL, -1,  wxT("Hello wxDC"),
     wxPoint(50,50), wxSize(WIDTH,HEIGHT))
    {
        sizer = new wxBoxSizer(wxHORIZONTAL);
        drawPane = new BasicDrawPane( this );
        sizer->Add(drawPane, 1, wxEXPAND);
        SetSizer(sizer);

        timer = new RenderTimer(drawPane);
        Show();
        timer->start();
    }
    ~MyFrame()
    {
        delete timer;
    }
    void onClose(wxCloseEvent& evt)
    {
        timer->Stop();
        evt.Skip();
    }
```

```
    DECLARE_EVENT_TABLE()
};


BEGIN_EVENT_TABLE(MyFrame, wxFrame)
EVT_CLOSE(MyFrame::onClose)
END_EVENT_TABLE()

bool MyApp::OnInit()
{
    frame = new MyFrame();
    frame->Show();

    return true;
}


BEGIN_EVENT_TABLE(BasicDrawPane, wxPanel)
EVT_PAINT(BasicDrawPane::paintEvent)
END_EVENT_TABLE()



BasicDrawPane::BasicDrawPane(MyFrame* _parent) :
wxPanel(_parent), parent(_parent)
{
  model = new SL::Model;
}
BasicDrawPane::~BasicDrawPane(){
  delete model;
}


void BasicDrawPane::paintEvent(wxPaintEvent& evt)
{
    wxPaintDC dc(this);
    render(dc);
}

void BasicDrawPane::paintNow()
{
    wxClientDC dc(this);
    render(dc);
}

```

```
void BasicDrawPane::render( wxDC& dc )
{
  //std::cout << this->model->forward(torch::ones({1,2})) << std::endl;
  this->model->train();
  auto size = this->parent->sizer->GetSize();
  dc.DrawBitmap(this->model->getMap().Rescale(size.GetX(), size.GetY()), wxPoint(0,0), false);
  dc.SetBrush(*wxTRANSPARENT_BRUSH);
  dc.DrawEllipse(
    size.GetX()/2 - size.GetX() * (double)CIRC_RADIUS/((double)WIDTH),
    size.GetY()/2 - size.GetY() * (double)CIRC_RADIUS/((double)HEIGHT),
    2 * size.GetX() * (double)CIRC_RADIUS/((double)WIDTH),
    2 * size.GetY() * (double)CIRC_RADIUS/((double)HEIGHT)
  );
}
```

6.1.5 include

6.1.5.1 point.hpp

```cpp
#pragma once
#include <iostream>
#include <wx/wx.h>

namespace VQ {
  class Point {
    public: using T = double;

    public: Point();
    public: Point(const Point& p);
    public: Point(const T& x, const T& y);
    public: Point(const std::pair<T, T>& pos);
    public: std::pair<T, T> getPos();
    public: const std::pair<T, T> getPos() const;

    public: void moveTo(const T& x, const T& y);
    public: void moveTo(const std::pair<T, T>& pos);
    public: void moveTo(const Point& point);
    public: void moveBy(const T& dx, const T& dy);
    public: void moveBy(const std::pair<T, T>& dpos);
    public: void moveBy(const Point& dpoint);

    public: Point operator*(const float scalar) const;
    public: Point operator*(const double scalar) const;
    public: Point operator-(const Point& other) const;
    public: Point operator+(const Point& other) const;
    public: Point& operator=(const Point&& other);
    public: Point& operator=(const std::pair<T, T>&& other);
    public: friend std::ostream& operator<<(std::ostream& os, const Point& dt);
    public: double abs() const;
    protected: std::pair<T, T> pos;

    public: void render(wxDC& dc, const wxBrush* brush) const;
  };

  Point randomPointInRange(const double& dx, const double& dy);
  std::ostream& operator<<(std::ostream& os, const Point& dt);
}
```

6.1.5.2 utils.hpp

```cpp
#pragma once

#include <cstdlib>
#include <vector>

namespace std {
  double rand(const double& max);
  size_t randMod(const size_t& mod);
}

namespace jce {
  template <typename T>
  T& randElement(std::vector<T>& vec);
  template <typename T>
  const T& randElement(const std::vector<T>& vec);
}

inline double std::rand(const double& max) {
  return (double)(std::rand()) / (((double)RAND_MAX/max));
}
inline size_t std::randMod(const size_t& mod) {
  return std::rand() % mod;
}

template<typename T>
T& jce::randElement(std::vector<T>& vec) {
  return vec[std::randMod(vec.size())];
}
template<typename T>
const T& jce::randElement(const std::vector<T>& vec) {
  return vec[std::randMod(vec.size())];
}
```

6.1.6 scr

6.1.6.1 point.cpp

```
#include <point.hpp>
#include <utils.hpp>
#include <config.hpp>

VQ::Point::Point(): pos({0,0}) {}
VQ::Point::Point(const VQ::Point& p): pos(p.getPos()) {}
VQ::Point::Point(const VQ::Point::T& x, const VQ::Point::T& y): pos({x, y}) {}
VQ::Point::Point(const std::pair<VQ::Point::T, VQ::Point::T>& _pos): pos(_pos) {}
std::pair<VQ::Point::T, VQ::Point::T> VQ::Point::getPos() { return pos;}
const std::pair<VQ::Point::T, VQ::Point::T> VQ::Point::getPos() const {return pos;}

void VQ::Point::moveTo(const VQ::Point::T& x, const VQ::Point::T& y) {
  this->moveTo(std::pair<double, double>(x, y));
}
void VQ::Point::moveTo(const std::pair<VQ::Point::T, VQ::Point::T>& _pos) {this->pos = _pos;}
void VQ::Point::moveTo(const Point& point) {
  this->moveTo(point.getPos());
}
void VQ::Point::moveBy(const VQ::Point::T& dx, const VQ::Point::T& dy) {
  this->pos = {
    dx + this->pos.first,
    dy + this->pos.second
  };
}
void VQ::Point::moveBy(const std::pair<VQ::Point::T, VQ::Point::T>& _pos){
  this->moveBy(_pos.first, _pos.second);
}
void VQ::Point::moveBy(const Point& point) {
  this->moveBy(point.getPos());
}

VQ::Point VQ::Point::operator*(const float scalar) const {
  return Point(
```

```
    scalar * this->pos.first,
    scalar * this->pos.second
  );
}
VQ::Point VQ::Point::operator*(const double scalar) const {
  return Point(
    scalar * this->pos.first,
    scalar * this->pos.second
  );
}
VQ::Point VQ::Point::operator+(const VQ::Point& point) const {
  return Point(
    this->pos.first + point.getPos().first,
    this->pos.second + point.getPos().second
  );
}
VQ::Point VQ::Point::operator-(const VQ::Point& point) const {
  return Point(
    this->pos.first  - point.getPos().first,
    this->pos.second - point.getPos().second
  );
}

VQ::Point& VQ::Point::operator=(const Point&& other) {
  return (*this = other.getPos());
}
VQ::Point& VQ::Point::operator=(const std::pair<VQ::Point::T, VQ
    ::Point::T>&& other) {
  this->pos = other;
  return *this;
}
double VQ::Point::abs() const {
  return std::sqrt(
      this->pos.first * this->pos.first
    + this->pos.second * this->pos.second
  );
}

void VQ::Point::render(wxDC& dc, const wxBrush* brush) const {
  dc.SetBrush(*brush);
  dc.DrawCircle(this->pos.first, this->pos.second, POINT_RADIUS)
    ;
}

VQ::Point VQ::randomPointInRange(const double& x, const double&
```

```
    y) {
  return Point(
    std::rand(x),
    std::rand(y)
  );
}
std::ostream& VQ::operator<<(std::ostream& stream, const VQ::Point& p) {
  stream << "<" << p.getPos().first << "< " << p.getPos().second << ">";
  return stream;
}
```

6.1.7 supervised

6.1.7.1 include

6.1.7.1.1 config.hpp

```
#pragma once

#define HEIGHT          500
#define WIDTH           500
#define RES_HEIGHT      200
#define RES_WIDTH       200
#define POINT_RADIUS    10
#define CIRC_RADIUS     200
#define BATCH_SIZE      512
#define EPOCHS          8
#define LR              0.1
#define Momentum        0.9
#define LOSS            torch::mse_loss
```

6.1.7.1.2 model.hpp

```cpp
#pragma once

#include <torch/torch.h>
#include <wx/bitmap.h>
#include <point.hpp>

namespace SL {
  class Model : public torch::nn::Module {
    private: using Optimizer          = torch::optim::SGD;
    private: using OptimizerOptions   = torch::optim::SGDOptions
    ;

    private: torch::nn::Linear lin1, lin2, lin3;
    private: torch::nn::LeakyReLU relu;
    private: torch::nn::Sigmoid sigm;
    private: Optimizer optim;

    public: Model();
    public: torch::Tensor forward(torch::Tensor x);
    public: void fit(torch::Tensor x, torch::Tensor y);
    public: void train();

    public: wxImage getMap();
    private: torch::Tensor getMapTensor();

    private: static std::pair<torch::Tensor, torch::Tensor>
    getTrainingData();
    private: static double classify(const VQ::Point& point);
  };
}
```

6.1.7.2 src

6.1.7.2.1 model.cpp

```cpp
#include <model.hpp>
#include <config.hpp>
#include <colormap/palettes.hpp>

#define COLORMAP "jet"

const static VQ::Point center(HEIGHT/2, HEIGHT/2);

SL::Model::Model() :
  lin1(register_module("lin1", torch::nn::Linear(2,256))),
  lin2(register_module("lin2", torch::nn::Linear(256, 16))),
  lin3(register_module("lin3", torch::nn::Linear(16, 1))),
  relu(),
  sigm(),
  optim(this->parameters(), OptimizerOptions(LR).momentum(Momentum))
{}

torch::Tensor SL::Model::forward(torch::Tensor x) {
  x = relu(lin1(x));
  x = relu(lin2(x));
  return sigm(lin3(x));
}

void SL::Model::fit(torch::Tensor x, torch::Tensor y_true) {
  torch::Tensor y_pred = this->forward(x);
  auto loss = LOSS(y_true, y_pred);
  loss.backward(loss);
  optim.step();
  optim.zero_grad();
}

void SL::Model::train() {
  auto data = this->getTrainingData();
  for (size_t idx=0; idx < EPOCHS; idx++)
    this->fit(data.first, data.second);
}

wxImage SL::Model::getMap() {
  torch::Tensor y = this->forward(this->getMapTensor());
  wxImage img(RES_HEIGHT, RES_WIDTH);
```

```
  auto pal = colormap::palettes.at(COLORMAP).rescale(0, 1);
  for (size_t idx_x=0; idx_x < RES_HEIGHT; idx_x++) {
    for (size_t idx_y=0; idx_y < RES_WIDTH; idx_y++) {
      auto pix = pal(y[idx_x*RES_WIDTH + idx_y][0].item<float>());
      img.SetRGB(idx_x, idx_y,
        pix.getRed().getValue(),
        pix.getGreen().getValue(),
        pix.getBlue().getValue()
      );
    }
  }
  return img;
}

torch::Tensor SL::Model::getMapTensor() {
  torch::Tensor out = torch::zeros({RES_HEIGHT * RES_WIDTH, 2});
  for (size_t idx_x=0; idx_x < RES_HEIGHT; idx_x++) {
    for (size_t idx_y=0; idx_y < RES_WIDTH; idx_y++) {
      out[idx_x*RES_WIDTH + idx_y][0] = (double)idx_x/((double)RES_HEIGHT);
      out[idx_x*RES_WIDTH + idx_y][1] = (double)idx_y/((double)RES_WIDTH);
    }
  }
  return out;
}

std::pair<torch::Tensor, torch::Tensor> SL::Model::getTrainingData() {
  torch::Tensor x = torch::ones({BATCH_SIZE, 2});
  torch::Tensor y = torch::ones({BATCH_SIZE, 1});

  for (size_t idx=0; idx < BATCH_SIZE; idx++) {
    const VQ::Point p = VQ::randomPointInRange(WIDTH, HEIGHT);
    y[idx][0] = classify(p);
    x[idx][0] = p.getPos().first  / (double)HEIGHT;
    x[idx][1] = p.getPos().second / (double)WIDTH;
  }
  return {x, y};
}
double SL::Model::classify(const VQ::Point& point) {
  if ((point - center).abs() > CIRC_RADIUS) {
    return 1.0;
  }
```

```
  return 0.0;
}
```

6.1.8 unsupervised

6.1.8.1 include

6.1.8.1.1 cluster.hpp

```cpp
#pragma once

#include <group.hpp>
#include <vector>

namespace VQ {
  class Cluster {
    private: std::vector<Point> points;
    private: std::vector<Group> groups;
    public:  Cluster(const size_t& points, const double& x, const double& y, const std::vector<const wxBrush*>& brushes);
    public:  void update();
    public:  void render(wxDC& dc);
    private: Group* getClosestGroup(const Point& point);
    private: const Group* getClosestGroup(const Point& point) const;
    private: const wxBrush* getPointBrush(const Point& point) const;
    private: void generatePoints(const size_t& count, const double& x, const double& y);
    private: void generateGroups(const double& x, const double& y, const std::vector<const wxBrush*>& brushes);
  };
}
```

6.1.8.1.2 config.hpp

```
#pragma once

#define HEIGHT          500
#define WIDTH           500
#define POINT_COUNT     100
#define POINT_RADIUS    5
#define GRPOUT_SIZE     15
```

6.1.8.1.3 group.hpp

```
#pragma once
#include <point.hpp>

namespace VQ {
  class Group: public Point {
    protected: const wxBrush* brush;

    public: Group();
    public: Group(const T& x, const T& y, const wxBrush* brush);
    public: Group(const std::pair<T, T>& pos, const wxBrush*
   brush);
    //public: ~Group();
    public: void render(wxDC& dc) const;
    public: const wxBrush* getBrush() const;
  };

  Group randomGroupInRange(const double& x, const double& y,
    const wxBrush* brush);
}
```

6.1.8.2 src

6.1.8.2.1 cluster.cpp

```cpp
#include <cluster.hpp>
#include <climits>
#include <float.h>
#include <utils.hpp>

VQ::Cluster::Cluster(const size_t& count, const double& x, const
    double& y, const std::vector<const wxBrush*>& brushes) {
  this->generatePoints(count, x, y);
  this->generateGroups(x, y, brushes);
}

void VQ::Cluster::update() {
  Point& point = jce::randElement(this->points);
  Group* group = this->getClosestGroup(point);
  Point delta = (point - (*group)) * 0.01;
  group->moveBy(delta);
}
void VQ::Cluster::render(wxDC& dc) {
  for (auto const point : this->points) {
    point.render(dc, this->getPointBrush(point));
  }
  for (auto const group : this->groups) {
    group.render(dc);
  }
}
VQ::Group* VQ::Cluster::getClosestGroup(const VQ::Point& point)
    {
  double distance = DBL_MAX;
  Group* out = nullptr;
  for (auto& group : this->groups) {
    double other = (group - point).abs();
    if (other < distance) {
      distance = other;
      out = &group;
    }
  }
  return out;
}
const VQ::Group* VQ::Cluster::getClosestGroup(const VQ::Point&
    point) const {
  double distance = DBL_MAX;
```

```
  const Group* out = nullptr;
  for (auto const& group : this->groups) {
    double other = (group - point).abs();
    if (other < distance) {
      distance = other;
      out = &group;
    }
    else {
      int a = 1;
    }
  }
  return out;
}
const wxBrush* VQ::Cluster::getPointBrush(const Point& point) const {

  return this->getClosestGroup(point)->getBrush();
}
void VQ::Cluster::generatePoints(const size_t& count, const double& x, const double& y) {
  this->points.clear();
  this->points.resize(count);
  for (auto iter = this->points.begin(); iter != this->points.end(); iter++) {
    *iter = randomPointInRange(x, y);
  }
}
void VQ::Cluster::generateGroups(const double& x, const double& y, const std::vector<const wxBrush*>& brushes) {
  this->groups.clear();
  this->groups.resize(brushes.size());
  for (size_t idx = 0; idx < brushes.size(); idx++) {
    groups[idx] = randomGroupInRange(x, y, brushes[idx]);
  }
}
```

6.1.8.2.2 group.cpp

```
#include <group.hpp>
#include <config.hpp>
#include <utils.hpp>

VQ::Group::Group(): Group(0,0,nullptr) {}
VQ::Group::Group(const VQ::Group::T& x, const VQ::Group::T& y,
    const wxBrush* _brush): VQ::Point(x, y), brush(_brush) {}
VQ::Group::Group(const std::pair<VQ::Group::T, VQ::Group::T>&
    pos, const wxBrush* _brush): VQ::Point(pos), brush(_brush) {}
//VQ::Group::~Group() { delete brush; }

VQ::Group VQ::randomGroupInRange(const double& x, const double&
    y, const wxBrush* brush) {
  return Group(
    std::rand(x),
    std::rand(y),
    brush
  );
}
void VQ::Group::render(wxDC& dc) const {
  dc.SetBrush(*(this->brush));
  dc.DrawRectangle(wxPoint(this->pos.first, this->pos.second),
    wxSize(GRPOUT_SIZE, GRPOUT_SIZE));
}
const wxBrush* VQ::Group::getBrush() const { return this->brush;
    }
```

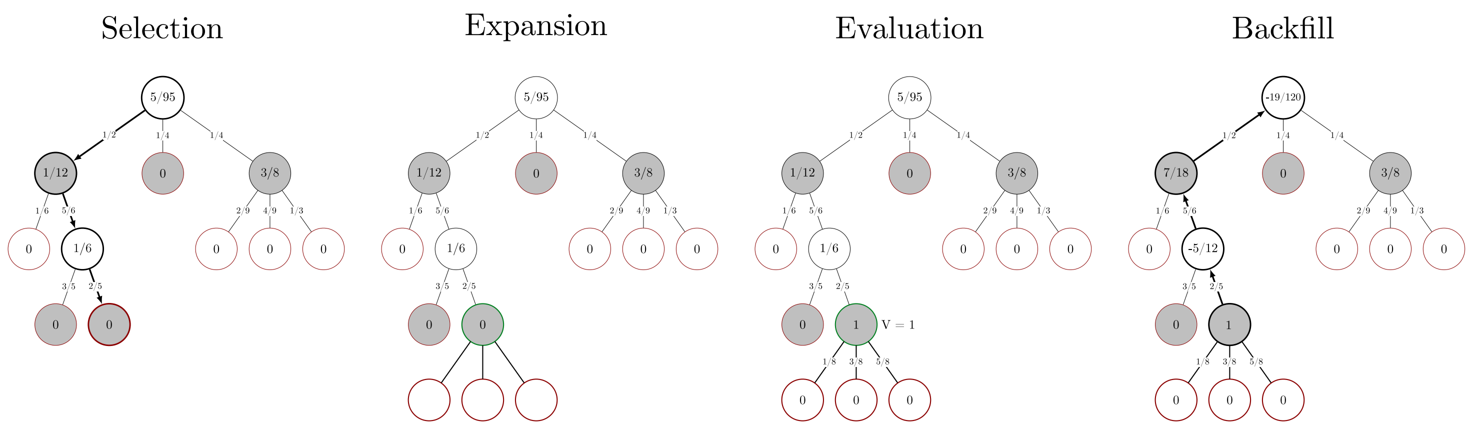


Figure 10: MCTS simulation steps. In this diagram, the numbers in the node represent Q and the number on the arrow is P. The red nodes are leaf nodes and the green one is the leaf node n_L. During the **selection** phase, σ is used to find successive nodes until the node n_L is reached. This is shown 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 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