ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΦΥΣΙΚΗΣ & ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΜΠΣ ΗΛΕΚΤΡΟΝΙΚΟΥ ΑΥΤΟΜΑΤΙΣΜΟΥ

ΜΕΤΑΠΤΥΧΙΑΚΗ ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ
ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ ΣΕ ΠΑΙΧΝΙΔΙΑ ΔΥΟ ΠΑΙΚΤΩΝ

ΙΑΣΩΝ Γ. ΓΑΒΡΙΗΛΙΔΗΣ

ΑΜ: 2013505

Επιβλέπων: Σταματόπουλος Παναγιώτης, Επίκουρος Καθηγητής

ΑΘΗΝΑ 2019

ΤΕΧΝΗΤΗ ΝΟΗΜΟΣΥΝΗ ΣΕ ΠΑΙΧΝΙΔΙΑ ΔΥΟ ΠΑΙΚΤΩΝ

ΙΑΣΩΝ Γ. ΓΑΒΡΙΗΛΙΔΗΣ
ΑΜ: 2103505

ΤΡΙΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

Σταματόπουλος Παναγιώτης, Επίκουρος Καθηγητής
Κοτρώνης Γιάννης, Αναπληρωτής Καθηγητής
Χατζηευθυμιάδης Ευστάθιος, Καθηγητής

ΕΥΧΑΡΙΣΤΙΕΣ

Θα ήθελα να εκφράσω τις θερμές ευχαριστίες μου σε όλους εκείνους που με βοήθησαν και με ενέπνευσαν κατά τη συγγραφή της παρούσας διπλωματικής εργασίας. Την ιδιαίτερη ευγνωμοσύνη μου θα ήθελα να εκφράσω στον επιβλέποντα καθηγητή μου, κύριο Σταματόπουλο Παναγιώτη για την ευκαιρία που μου έδωσε να καταπιαστώ ερευνητικά με ένα τόσο ενδιαφέρον αντικείμενο, το οποίο ανταποκρίνεται πλήρως στα επιστημονικά μου ενδιαφέροντα. Με την εμπιστοσύνη που μου έδειξε, τη συνεχή υποστήριξή του, την υπομονή του και τις ουσιαστικές παρατηρήσεις του σε όλη τη διάρκεια εκπόνησης, συνέβαλε ουσιαστικά στη συγγραφή και ολοκλήρωση της εργασίας. Επίσης, ευχαριστώ τα υπόλοιπα μέλη της τριμελούς επιτροπής, τον κύριο Κοτρώνη Γιάννη και τον κύριο Χατζηευθυμιάδη Ευστάθιο, για τη συμμετοχή τους στην αξιολόγηση της διπλωματικής μου εργασίας.

ΠΕΡΙΛΗΨΗ

Η παρούσα διπλωματική εργασία έχει σκοπό να διερευνήσει μια σειρά από μεθόδους τεχνητής νοημοσύνης στον τομέα των παιχνιδιών δύο παικτών. Για την επίτευξη αυτού του σκοπού διεξήχθησαν πειράματα σε δύο παιχνίδια, στο Othello και στο Quoridor, χρησιμοποιώντας τρεις αλγορίθμους. Συγκεκριμένα, οι αλγόριθμοι που χρησιμοποιήθηκαν ήταν ο Alpha-Beta, ο Monte Carlo Tree Search και ο Alpha-Zero. Τα ευρήματα των πειραμάτων ανέδειξαν την θεμελιωμένη πλέον θέση των παραδοσιακών αλγορίθμων Alpha-Beta. Επιπλέον, αναδείχθηκε η ισχύς του Alpha-Zero να εξάγει αποτελέσμτα ανεξαρτήτως παιχνιδιού και επιβεβαιώθηκε σε θεωρητικό επίπεδο ότι είναι δυνατό να ξεπεράσει κάθε άλλον αλγόριθμο. Τέλος, ο Monte Carlo Tree Search παρότι δεν είναι δυνατό να πάραξει ανταγωνιστικά αποτελέσματα μελετήθηκε ώστε να υπάρχει επιστημονικά μια συνέχεια στην αλγοριθμική σκέψη.

Λέξεις-κλειδιά: παιχνίδια δύο παικτών, τεχνητή νοημοσύνη, νευρωνικά δίκτυα

ABSTACT

This dissertation aims at exploring a number of artificial intelligence methods in the field of two player games. In order to achieve this goal several experiments have been conducted in two particural games, Othello and Quoridor, utilising three algorithms. Specifically, Alpha-Beta, Monte Carlo Tree Search and Alpha-Zero algorithms have been used. The results of the experiments have highlighted the established position of the Alpha- Beta traditional algorithms. Moreover, the power of Alpha- Zero algorithm has been underpinned and it is suggested that this algorithm could transcend any other algorithm. Finally, although Monte Carlo Tree Search can not provide competitive results, however it has been studied so that there is scientific continuity in algorithmic thinking.

Keywords: two player games, artificial intelligence, neural networks.

Contents

Chapter 1

Introduction

Two player games is an important domain of artificial intelligence. The only knowledge needed to be provided is the rules of the game (along with the valid moves) and the terminal conditions (winning/losing states of the game). Since both players objective is to win the game, searching for solutions in such space is called Adversarial Search. Building good agents is a difficult task due the number of possible states a game can have. Games can be represented as a tree with nodes as states and edges as actions connecting an initial and a resulting state.

Traditional algorithms are trying to search through the game tree to generate solutions. The branching factor and the depth of the tree, make a complete search impossible, therefore in order to narrow the search space, domain specific evaluation functions need to be constructed. Evaluation functions are designed so that a non terminal state can be evaluated for its performance with needing to traverse the underlying subtree. Reduction of the search space can be also achieved by identifying subtrees that are unable to produce better results that the ones already explored, and exclude them from the search. The previous characteristic can be utilized so that a search firstly discovers the better performing subtrees, which in consequence results in bigger subtrees beeing eliminated from the search process. Finally, computation can be reduced by reusing results from a database of known states, where this database can be either generated during search or provided externally. The methods and approaches presented above are utilized by the Alpha-Beta algorithm.
Traditional algorithms have been proven to be able to produce really strong players. The biggest downside is the innability to generalize, since the evaluation functions are domain dependent. A domain independent algorithm cannot depend on evaluation functions, and as a result limiting the search space is not an option. By simulating complete games from a non terminal state and leveraging statistics we can gather probabilistic results for the outcome of a game. The results can be later used to guide the play of an agent. This method is domain independent, in the sense that only the rules of the game need to be supplied, and it's used by Monte Carlo Tree Search algorithm with interesting results.

The idea behind simulating games and gathering probabilistic results is quite powerful, however it cannot lead to strong plays in reasonable time since all the effort made is lost after the game. The rise of technology and the evolution of neural networks, have provided a new algorithm that combines the domain independent properties of Monte Carlo Tree search with the properties of neural networks. The weights of a neural network could be considered as a compressed database of results. Training a neural network with the results generated from Monte Carlo Tree Search, can later replace the search itself with the neural network. This is a pioneer idea that has been used in the Alpha-Zero algorithm, and it has provided excellent results when competing other algorithms.

Chapter 2

Adversarial Search

One of the corner-stones of Artificial Intelligence is problem solving. In the context of problem solving, we can identify two independent processes: representing knowledge and performing search. In this chapter we introduce the formulation for two-player, zero-sum, perfect information games and present the foundation for search algorithms.

2.1 Search Problems

Intelligence is concerned mainly with rational action. We use the notion of intelligent agent to describe an entity which makes the best possible action in a given situation. The study of problem in Artificially Intelligence is mainly to build agents that are intelligent in this sense.

Rational action for search problems should take account some sort of goal information which describes a state that is desirable. Agents performing on environments where current state is insufficient and goal information is needed are called Goal-based agents.

In order for a Goal-based agent to maximize its performance the following steps should be considered:

- **Goal Formulation**: Organize behavior by limiting the objectives the agent tries to achieve.
- **Problem Formulation**: Define the states/actions need to be considered, given a goal.
- **Search**: The process of finding a sequence of actions for a problem that achieves it's goal.
- **Execute**: Given a solution returned from search process, each action is carried out.

A problem can be formally represented[11] by the following components:

- **Initial State**: The state that the agent starts with.
- **Actions**: A description of all the possible actions available to the agent. Given a state s, $Actions(s)$ returns all applicable actions that can be executed in s.
- **Transition Model**: A description of the result of each action applied on its applicable states. Given a state s and an action a, $Result(s, a)$ returns the state that results from applying action a in state s. We use the term *successor* to refer to any state reachable from any given state by a single action.
- **Goal Test**: A function that determines whether a given state is a goal state.
- **Path Cost**: A function that assigns a cost for each path in the State Space. Path Cost is the sum of the costs for each action taken along the path. The cost of an action a in a state s is called *stepcost*.

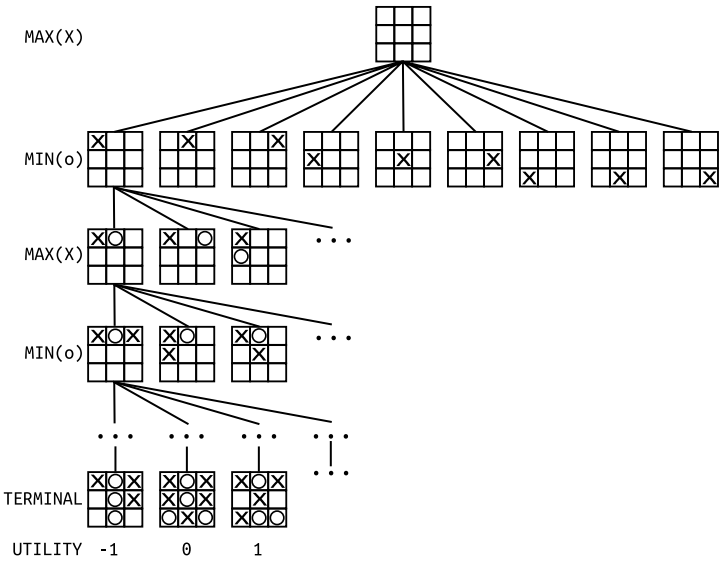
The composition of Initial State, Actions and Transition Model is defined as the **State Space** - the set of all states reachable from the initial state by any sequence of actions. The State Space of a Problem forms a **graph** where nodes are the states and edges between nodes are the actions. Given the above representation a *path* in the State Space is a sequence of states connected by actions.

The environments that we are interested in, are multi-agent environments where each agent needs to consider the actions taken by other agents in order to decide on how to maximize its performance. Such environments where agents goals conflict, are competitive and give rise to a specific set of search problems - **adversarial search** problems - also known as games.

Games are a special kind of search problems so they can be represented in a similar manner. Since we mostly interested in two-player games, we choose MIN and MAX for players names. The game is played in rounds and at the end of it, points are awarded to the winning player and penalties are given to the looser. A game can be formally represented[11] by the following components:

- **Initial State**: The initial setup of the game.
- **Player**: Defines who's player turn is for a given state. Given a state s, $Player(s)$ returns which player has the move.
- **Actions**: A description of all the legal moves in a given state. Given a state s, $Actions(s)$ returns all applicable actions that can be executed in s.
- **Result**: A description of the result of a move. Given a state s and an action a, $Result(s, a)$ returns the state that results by applying move a in state s.
- **Terminal Test**: A function that determines whether the game is over. Multiple states where the game has ended may exist and they are called terminal states.
- **Utility**: Defines the final numeric value for a game that ends in a state s from a player s. $Utility(s, p)$ may take different values for different games. $Zero - sum$ games are defined as games where the total utility value for all players is independent of the instance of the game.

The composition of Initial State, Actions function and Result function is defined as the **Game Tree** - a tree where nodes are game's states and edges are the moves. Players execute their move until leaf nodes, nodes corresponding to terminal states, are reached. The number on each leaf node indicates the utility value of the terminal state from the point of view of MAX player. The term **Search Tree** is used to indicate a subset of the game tree and examines enough moves in order to allow a player to decide on his next move. A visual example of tick-tack-toe is presented in order to give a better understanding for the representation of a Game Tree:



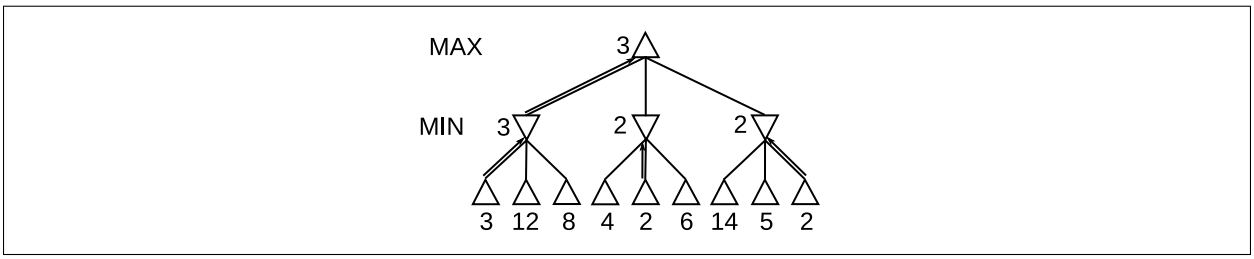
An expanded game tree for the game of tic-tac-toe. The game tree contains all the intermediate states from the initial state to all possible terminal states.

2.2 Minimax Algorithm

Games unlike other search problems are hard to solve. This is because game trees usually have a big branching factor and depth, which result in large game trees. Searching for optimal solutions in large trees is infeasible, so other strategies should be explored.

An optimal solution for a game would be a series of moves leading to a goal state (i.e. a terminal state where MAX player is the winner). Given a complete game tree, the optimal strategy can be determined for **minimax value**[14] (evaluation function) of each node of the game tree which we express as $Minimax(n)$. MAX player will always prefer to make a move resulting in a state of maximum value, whereas MIN will prefer a move resulting in state of minimum value.

$$Minimax(s) = \begin{cases} Utility(s) & if \;\; TerminalTest(s) \\ max_{a \in Actions(s)} Minimax(Result(s,a)) & if \;\; Player(s) == MAX \\ min_{a \in Actions(s)} Minimax(Result(s,a)) & if \;\; Player(s) == MIN \end{cases} \tag{2.1}$$



The **minimax algorithm** uses the above definition in order to recursively compute the minimax values for each successor of a state. The recursion proceeds until it reaches the leaves of the game tree and minimax values are **backed up**.

Since the algorithm is performing a complete depth-first exploration of the game tree, for a tree of depth m where there are b legal moves for each state the time complexity is $O(b^m)$. The space complexity is $O(b \cdot m)$ for an algorithm that generates all moves at once, or $O(m)$ for an algorithm that generates one move at a time (backtracking algorithm). It's clear that time cost is impractical for real games, so this algorithm serves the basis for mathematical analysis of games.

Algorithm Minimax decision

```
function MINIMAX(state)
    action ← argmax_{action∈ACTIONS(state)} MINVALUE(Result(state, action))
    return action
end function

function MAXVALUE(state)
    if TERMINAL-TEST(state) then
        return UTILITY(action)
    end if
    value ← −∞
    for all action ∈ ACTIONS(state) do
        value ← max(value, MINVALUE(Result(state, action)))
    end for
    return value
end function

function MINVALUE(state)
    if TERMINAL-TEST(state) then
        return UTILITY(action)
    end if
    value ← +∞
    for all action ∈ ACTIONS(state) do
        value ← min(value, MAXVALUE(Result(state, action)))
    end for
    return value
end function
```

2.3 Depth Limiting - Approximate Evaluation

Perfect decisions are possible only in principle for real games - exploring the complete game tree is unfeasible. Our goal is to develop a strategy that will be able to perform relatively good (compared to a human) with significant reduced time complexity needs.

We introduce the notion of heuristic **evaluation function** which is based on the general structure of a state. We can now limit the depth of the search by effectively turning non-terminal nodes into terminal states[14]. We enhance the game representation with the following functions:

- **Evaluation function**: Defines a heuristic value for a state which estimates the position's utility value.
- **Cutoff Test**: A function which decides when to apply the evaluation function on a state and break the recursion.

$H \cdot Minimax(s, d) =$

$$\begin{cases} Eval(s) & if \;\; CutoffTest(s, d) \\ max_{a \in Actions(s)} H \cdot Minimax(Result(s, a), d+1) & if \;\; Player(s) == MAX \\ min_{a \in Actions(s)} H \cdot Minimax(Result(s, a), d+1) & if \;\; Player(s) == MIN \end{cases} \quad (2.2)$$

The requirements of an Evaluation function are:

- it should order terminal states in a similar manner Utility function does: states that win must evaluate better than states that lose.
- non-terminal states should be strongly correlated with the actual chances of winning/losing the game.
- the computation time required for the execution should be as low as possible in order to be able to reduce time complexity of search.

Evaluation functions are usually a composition of feature calculations of the state. The most common implementation is based on combination of the numerical contributions for each feature.

These function are called **weighted linear functions** and can be expressed as:

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

where w_i is the weight each feature contributes and f_i is the feature value of the position. Combining features in a linear fashion assumes that each feature is independent from the values of other features. This assumption may not be a good fit in some games, so nonlinear combinations for evaluation functions are also possible.

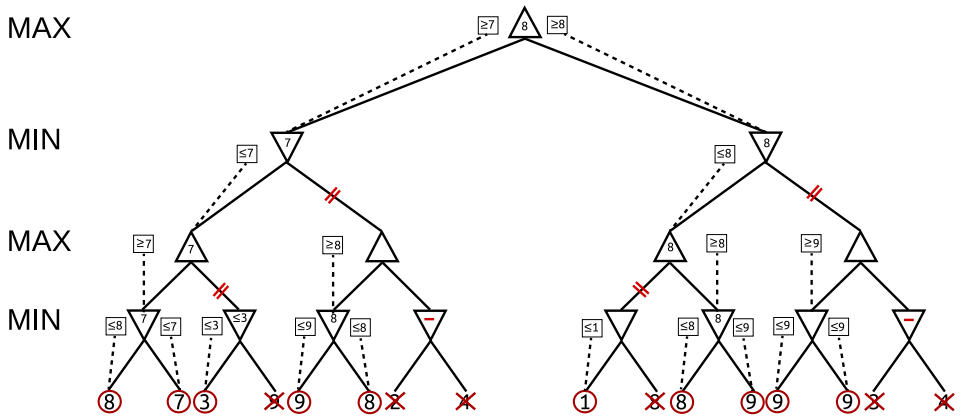
Chapter 3

Alpha-Beta Pruning

The efficiency with which large game trees can be searched is of critical importance. As noted Minimax algorithm explores the whole tree so examined nodes are exponential regards to the depth of the tree. In this section, we examine **alpha-beta pruning** technique which can reduce the examined nodes by half. The basic idea behind this technique is that it is possible to compute the correct minimax values without exploring every node in the game tree.

3.1 Alpha-Beta Algorithm

Alpha-Beta algorithm reduces the time complexity for a complete depth-first exploration of the game tree by pruning branches of the tree that not affect the backed up value. The following example -for a game tree that maximizes at top level- gives a visual example of how these branches are detected:



A detailed visualization for the alpha-beta algorithm. The exploration of the game tree is reduced by detecting cut-offs (shallo or deep). At each node the value backed-up from the child nodes is noted with a square, nodes marked with an x are ignored due to cut-offs and the subtrees affected by cut-offs are denoted with a dashed line.

The principal behind pruning is the following: consider a node n where player (MIN or MAX) has the choice of moving on it. If the player has a better choice m either at the parent node of n or any ancestor of the parent node, then n will never be reached in the actual game. Each node is assigned with α and β

values, which correspond to lower and upper limit for the backed-up values from successor nodes. These two parameters have given the name for the algorithm and are used to determine if a better choice is observed along the search path for MAX and MIN player:

- α: the value of the best (highest) choice found so far at any visited node along the path for MAX player.
- β: the value of the best (lowest) choice found so far at any visited node along the path for MIN player.

The algorithms operates by updating α and β values for each node visited, and prunes the remaining branches as soon as the value of the current node is known to be worse than the current α of β value (for MAX and MIN respectively).

Algorithm Alpha Beta

```
function AlphaBeta(state)
    v ← Maxvalue(state, −∞, +∞)
    return action ∈ Actions(state) where value = v
end function

function maxvalue(state, α, β)
    if Terminal-test(state) then
        return Utility(action)
    end if
    value ← −∞
    for all action ∈ Actions(state) do
        value ← max(value, Minvalue(Result(state, action)))
        if value ≥ β then
            return value
        end if
        α ← max(α, value)
    end for
    return value
end function

function minvalue(state, α, β)
    if Terminal-test(state) then
        return Utility(action)
    end if
    value ← +∞
    for all action ∈ Actions(state) do
        value ← min(value, Maxvalue(Result(state, action)))
        if value ≤ α then
            return value
        end if
        β ← min(β, value)
    end for
    return value
end function
```

A formal definition [12] of alpha-beta pruning algorithm is given below:

- each node at depth $d \leq D$ is identified as $p(\vec{i_d})$, where $\vec{i_d}$ is a vector of length d whose components $i_1, i_2, ..., i_d$ identify the branch selected from the nodes at successive depths in the tree along the path from root node to $p(\vec{i_d})$.
- $v(\vec{i_d})$ is the backed-up value (for an intermediate node) or the utility value (for a leaf node) associated with the node $p(\vec{i_d})$.

We introduce the following notation:

$$[k]_e = 2[\frac{k}{2}] = \begin{cases} k & if \quad k \quad is \quad even \\ k-1 & if \quad k \quad is \quad odd \end{cases}$$

$$[k]_o = 2[\frac{k-1}{2} + 1] = \begin{cases} k & if \quad k \quad is \quad odd \\ k-1 & if \quad k \quad is \quad even \end{cases}$$

Examining the path from the root to the node $p(\vec{i_d})$ at depth j, where j and d are even with $0 \leq j < d$, a maximizing operation is in progress. The lower bound $a_j(\vec{i_d})$ on $v(\vec{i_j})$ for the value of node $p(\vec{i_d})$ is denoted as the **j-depth alpha value**, where:

$$a_j(\vec{i_d}) = \begin{cases} max\{v(i_1,...,i_j,1), v(i_1,...,i_j,2), v(i_1,...,i_j,i_{j+1}-1)\} & for \;\; i_{j+1} > 1 \\ -\infty \;\; for \;\; i_{j+1} = 1 \end{cases}$$

Similarly,examining the path from the root to the node $p(\vec{i_d})$ at depth j, where j and d are odd with $0 \leq j < d$, a minimizing operation is in progress. The upper bound $b_j(\vec{i_d})$ on $v(\vec{i_j})$ for the value of node $p(\vec{i_d})$ is denoted as the **j-depth beta value**, where:

$$b_j(\vec{i_d}) = \begin{cases} min\{v(i_1,...,i_j,1), v(i_1,...,i_j,2), v(i_1,...,i_j,i_{j+1}-1)\} & for \;\; i_{j+1} > 1 \\ +\infty \;\; for \;\; i_{j+1} = 1 \end{cases}$$

Having the definitions at j-level for alpha and beta value set, we can define **alpha value** (i.e. the highest alpha value) and **beta value** (i.e. the lowest beta value) as:

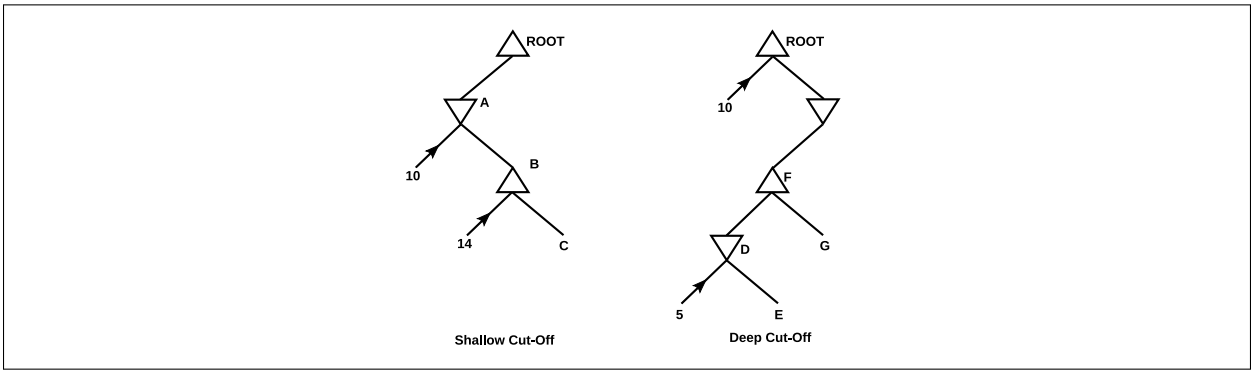
$$a(\vec{i_d}) = max\{a_0(\vec{i_d}), a_2(\vec{i_d}), ..., a_{[d]_e}(\vec{i_d})\}$$

$$b(\vec{i_d}) = min\{b_1(\vec{i_d}), b_3(\vec{i_d}), ..., b_{[d]_o}(\vec{i_d})\}$$

Given the above definitions we can deduce that if at depth k the maximum value of $a_j(\vec{i_d})$ is attained, then the highest alpha value is a lower bound of the eventual back-up value of the sub-tree rooted at $p(\vec{i_k})$ and the exploration sub-trees, whose back-up value cannot be greater, can be skipped.

The alpha-beta pruning algorithm is identical to the minimax algorithm except that:

$$\left| \begin{array}{l} v(\vec{i_d}) \leq a(\vec{i_d}) \;\; , d \;\; even \Rightarrow alpha \;\; cutoff \;\; occurs \\ v(\vec{i_d}) \geq b(\vec{i_d}) \;\; , d \;\; odd \;\; \Rightarrow beta \;\; cutoff \;\; occurs \end{array} \right|$$



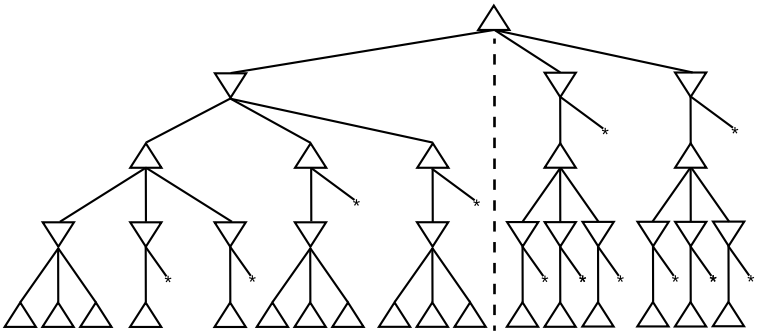
If $v_{ab}(\vec{i_0})$ is the backed-up value of a game tree using the alpha-beta pruning algorithm and $v_{mm}(\vec{i_0})$ is the backed-up value of the same game tree using the minimax algorithm, then:

$$v_{ab}(\vec{i_0}) = v_{mm}(\vec{i_0})$$

The lower limit for the number of leaf nodes examined [16] for a game tree with depth d and constant branching factor b is:

$$N_d = \begin{cases} 2b^{\frac{d}{2}} - 1 & d \;\; even \\ b^{\frac{d+1}{2}} + b^{\frac{d-1}{2}} & d \;\; odd \end{cases}$$

The lower limit is achieved by *perfect ordering* at every level so that all alpha and beta cutoff occurs. Heuristics for reordering the nodes towards the "perfect" arrangement, that correlate the static value of an intermediate node to the backed up value for the leaf nodes, can achieve a better performance for alpha beta search.



Perfect ordering of a game tree where cut-offs occur on every level of the tree.

The characteristics of alpha beta algorithm in terms of performance justify why this is the most widely used algorithm for two player games.

3.2 Alpha-Beta Search Enhancements

3.2.1 Iterative deepening

Iterative deepening is a build up on idea to perform search in a depth first manner by iterative increasing the depth of the search in a tree. In order to understand the benefits of iterative deepening, we will compare it with the standard depth-first and breadth-first search techniques, it terms of time and space complexity. [7]

We already saw that the time complexity of a depth-first exploration is $O(b^d)$ and the space complexity is $O(d)$ for an algorithm that generates one move at a time (backtracking algorithm). Breadth-first is expanding all state to a certain depth before continuing deeper. In the worst case all states must be generated up to depth d, or $b + b^2 + b^3 + ... + b^d$ which is $O(b^m)$. Since all the nodes at a given depth are stored in order to generate the nodes at the next depth, the minimum number of nodes that must be stored to search to depth d is b^{d-1}, which is $O(b^d)$.
Since Iterative Deepening expands all nodes in a given depth before continuing to a deeper depth the space complexity is $O(d)$. Its disadvantage is that it wastes computation before reaching to the goal depth. Nodes at depth d are generated once during the last iteration of the search, nodes at depth $d-1$ are generated twice etc. This makes the total number of generated nodes for a search at depth d:

$$b^d + 2b^{d-1} + 3b^{d-2} + ... + db = b^d(1 + 2b^{-1} + 3b^{-2} + ... + db^{1-d})$$

We know that for $abs(x) < 1$ the series $1 + 2x^1 + 3x^2 + 4x^3 + ...$ converges to $b^d(1 - x^2)$. Since the branching factor $b > 1$ the time complexity of iterative deepening is $O(b^d)$. The space complexity since we are following a depth first search will be $O(d)$. The advantages of iterative deepening depth first search are:

- It is guaranteed to find the optimal path
- Exploration is limited by the depth so it does not explores the whole tree
- The memory complexity lower that the Breadth first search.

3.2.2 Transposition tables

It is common in games the same positions to be reached from different combinations of moves. When searching a game tree, computation can be saved by caching results of states and avoid re-computation when found again. The idea is to trade computational time against memory space and this is achieved by the use of Transposition tables [9]. There are different approaches to determine which states to cache in order to achieve higher cache hit in the table. Transposition tables can be used in two manners: if the position is found and has been searched to the desired depth, the value of the node can be directly retrieved from the table and if the position is found but the information is not reliable enough, because of the depth of the search, the move should be tried first, as there is high probability to be also the best move in the current depth. Since a Transposition Table has finite capacity, a replacement policy can be used to manage this limited capacity, i.e., determine how/which entries are retained or replaced when the table is full. A significant performance improvement can be achieved with a special hashing algorithm, called Zobrist hash[17], that is specially designed for game boards. This algorithm will not compute the hash based on the entire board but update it using the hash of the previous board and applying a XOR function with the changed positions.

3.2.3 Minimal Window

The idea behind minimal window[13] is to reduce the size of the game tree by increasing the likelihood of a cutoff. In order to achieve this after evaluating the first move the remaining moves are search with a window of $(-\alpha - 1, -\alpha)$ instead of $(-\beta, -\alpha)$. This way we can quickly decide if one of the remaining moves leads to an inferior variation. If the value returned falls into the alpha-beta window, the sub-tree must be search with the $(-\beta, -\alpha)$ window in order to determine the correct value of the node.

3.2.4 Killer Heuristic

The killer heuristic involves storing moves (also called killers) at each depth which seem to be causing the most cutoffs. Next time the same depth in the tree is reached, the killer move is retrieved and used, if valid in the current position. Searching the best move first has a dramatic effect on tree size so it is important to have a good move ordering. In positions where transposition table can be used, the best move can be suggested based on information acquired from previous searches. Ordering the remaining moves is a difficult task and requires domain specific heuristic knowledge.

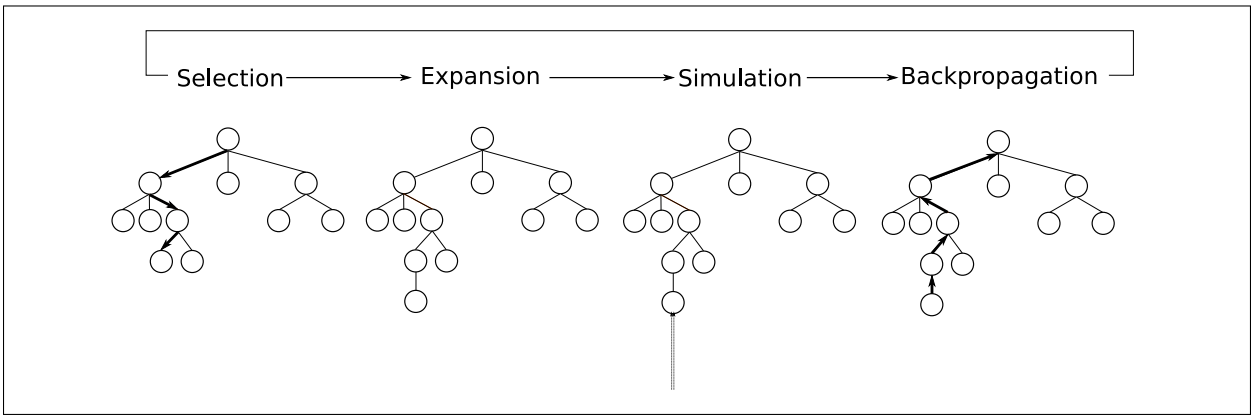
3.2.5 History Heuristic

History heuristic also stores moves that seem to be causing the most cutoffs but calculations are retrieved only for identical but also for similar positions. The intuition behind this approach is that minor differences in positions may not alter a move from still being the best. History heuristic maintains good moves for all depths throughout the search.

Chapter 4

Monte Carlo tree search

Depth limiting is a crucial part of traditional search algorithms. Building good estimates for non terminal game states is a complex domain-dependent task which is difficult to achieve. Monte Carlo tree search is a technique that uses stochastic simulations, it does not require domain specific knowledge and can be applied to any game with finite length.



4.1 MCTS Algorithm

The basis of the algorithm is the simulation of games where both players select their actions randomly. This does not seem like a good option, but when simulating a large number of games, a good strategy can be inferred[1] [2]. A tree of possible future games is built up iterative and used in order to probabilistic determine the best action for a given state.

- **Selection** While the state is found in the build-up tree, the next action is to select according to statistics stored (the number of times this state has been visited, sum of the values backed up from random games). The algorithm does not blindly selects actions with better probabilities, but instead selects in a fashion that keeps balance between **exploration** and **exploitation**. Exploration refers to selecting less promising actions that haven't been excessively explored and exploitation refers to selecting actions that lead to the best results (with the current knowledge)
- **Expansion** When the state cannot be found in the build-up tree, it is added in the tree with empty statistics. The tree is expanded after each simulated game.

- **Simulation** The game from this point and on (until a terminal state) is played by selecting actions uniformly at random for both players. Heuristic knowledge can enhance this process in order to improve the probabilities of selecting a more promising action.
- **Back-propagation** After reaching the end of the simulated game, we update the statistics of each tree node that was traversed during that game. The visit counts are increased and the win/loss ratio is modified according to the outcome.

The action from the current state that is selected after the specified number of iterations, is the most promising as defined in the selection process.

Algorithm Monte Carlo Tree Search

```
function MCTS(state)
    u_0 ← Node(state)
    while within budget do
        u_i ← TreePolicy(u_0)
        Δ ← DefaultPolicy(state(u_i))
        Backup(u_i, Δ)
    end while
    return action(BestChild(u_0))
end function
```

The four steps in the algorithm can be grouped in two distinct policies:

- **Tree Policy**: Select or create a leaf node from the nodes already contained within the search tree (selection and expansion).
- **Default Policy**: Play out the domain from a given non-terminal state to produce a value estimate (simulation).

The back-propagation step does not use a policy, but updates node statistics that affect future tree policy decisions.

4.1.1 Selection

The selection step is based on a selection strategy that is applied recursively until a node that is not part of the tree is reached. This strategy controls the balance between exploitation and exploration. Similar balancing problems have been studied with respect to the Multi-Armed Bandit problem. Multi-armed bandit setting[10] is a simple mathematical model for sequential decision making in unknown random environments that illustrates the so-called **exploration-exploitation trade-off**.

Given K arms with reward distributions $\theta_k \in [0, 1]$ (initially unknown to the user), the multi-armed bandit problem can be defined by a tuple of $< A, R >$ where:

- At each step t, we take an action a on one arm and receive a reward r.
- A is the set of actions, each referring to an interaction with an arm. The value of an action a is the expected reward $Q(a) = E[r|a] = \theta$. If an action a_t at step t is on the i_{th} arm then the $Q(a_t) = \theta_i$
- R is the reward function. At step t, $r_t = R(a_t)$ may return reward 1 with probability $Q(a_t)$ else 0.

The goal of the agent, is to maximize the cumulative reward $\sum_{t=1}^{T} r_t$ or equivalently to minimize the regret (loss). Since the distributions of the rewards are initially unknown, several pulls to each of the arms are needed, in order to acquire information(exploration). While the knowledge improves, pulls to the apparently best arms are increased (exploitation).

The selection process can be view as a Mutli-Armed Bandit problem with the difference that it operates recursively by applying multiple selections. While some strategies are directly originated for Multi-Armed Bandit problem others have been specifically designed for MCTS.

- **OMC**[4] Objective Monte Carlo consists of an urgency function $U(a)$ for each available action a and a fairness function that decides on the action with proportion to its urgency.

$$U(a) = erfc(\frac{u_0 - u_a}{\sqrt{2}\sigma_a})$$

where u_0 is the value of the best action, u_a and σ_a the value and the standard deviation of the move under consideration and erfc is the complementary error function:

$$erfc(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{u^2} du$$

The selection strategy then selects the node i that maximized the fairness function:

$$f_i = \frac{n_{parent(i)} U(i)}{n_i \sum_{j \in S_i} U(j)}$$

Where n_i is the visit count of the node i, and s_i is a set containing all the sibling nodes of i.

- **PBBM** [5] Probability to be Better that Best Move is similar to OMC as it consists of an urgency function $U(a)$ that is proportional to the probability of the move to be better that the current best move. PBBM takes account the standart deviation of the best move:

$$U(a) = exp(-2.4 \frac{u_0 - u_a}{\sqrt{2(\sigma_0^2 + \sigma_a^2)}})$$

where u_0, σ_0 are the value and the standard deviation of the best action, u_a and σ_a the value and the standard deviation of the move under consideration.

- **UCT** [6] Upper Confidence bounds is a strategy directly inherited from Mutli-Armed Bandit Problem [8]. It is the most common strategy used in MCTS as it is easy to implement and achieve good results. Let I to be all the set of all reachable nodes for the current tree node n. The uct selects the child c that maximizes the Upper Confident Bound:

$$c = argmax_{i \in I}(u_i + C\sqrt{\frac{\ln n_{parent(i)}}{n_i}})$$

Where n_i is the visit count of the node i and u_i is the value of the node i. Controlling the preference of exploitation over exploration is achieved by tuning the paramenter C.

4.1.2 Expansion

The expansion step adds nodes to the tree. Since even in the most simplistic domains the whole game tree cannot be stored in memory, an expansion strategy decides, given a node, if it will be expanded by storing one or more of its children in memory. The *max* expansion strategy adds one node per simulated game. This node corresponds to the first position encountered during the traversal that was not already stored in memory. Other strategies have been proposed that either expand the tree to a certain depth or forbid expansion before a certain amount of simulation has been reached. Generally, the effect of different strategies on MCTS doesn't affect the performance which is the reason why one expansion per simulation is preferred.

4.1.3 Simulation

The simulation step selects moves in a self-playing game until a terminal state has been reached. Actions can be either selected randomly or chosen according to a simulation strategy. The simulation strategy is also subject to trade-offs. Adding heuristic knowledge can improve the performance of the algorithm by

guiding to the most interesting actions. On the other hand when simulating using randomness, the tree search will be deeper since simulation can be faster compared to simulations using heuristic knowledge. The exploitation-exploration trade off comes in play here too. If the simulation is stochastic, the exploration is preferred causing unrealistic playouts. If the simulation is deterministic, the exploitation is preferred causing the search space to become selective.

4.1.4 Backpropagation

The backpropagation step propagates back the result of a game from a leaf node (terminal state) to nodes that had to be traversed in order this leaf node to be reached. For zero-sum games the result can be either $+1$ for won games and -1 for lost games. The most common strategy is averaging the results to compute the value of a node.

4.1.5 Move Selection

After running multiple simulations, the move that is selected is the *best child* of the root node. There are several ways to define *best*:

- **Max** The child with the highest value.
- **Robust** The child with the highest visit count.
- **Robust-max** The child with the highest value and visit count. If no child can be found that satisfies both, more simulations are played until one is found.
- **Secure** The child that maximizes the lower confidence bound: $u + \frac{A}{\sqrt{n}}$ where u is the node's value, n is the node's visit count and A a tuning parameter.

The algorithm used for evaluation is using UCT for the selection process, uniform random moves for expansion and Robust child for move selection. The complete algorithm is the following:

Algorithm UCT

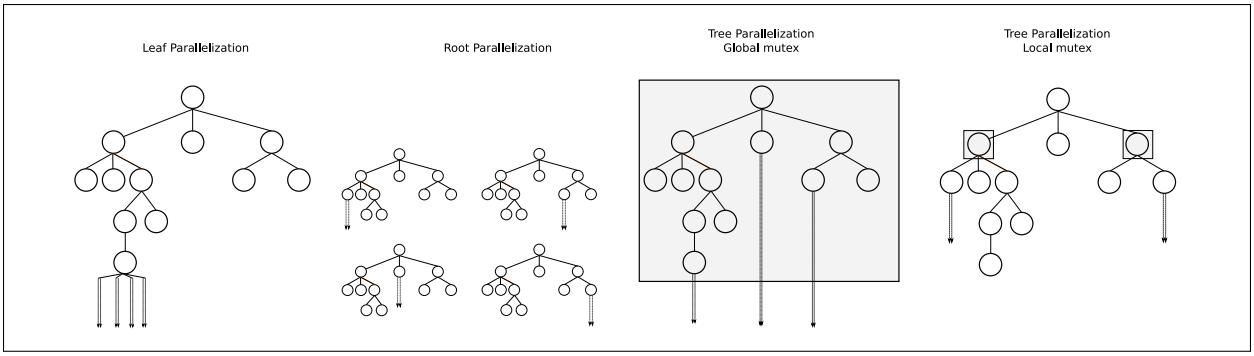
```
function UctSearch(state)
    u_0 ← Node(state)
    while within budget do
        u_i ← TreePolicy(u_0)
        Δ ← DefaultPolicy(state(u_i))
        Backup(u_i, Δ)
    end while
    return action(BestChild(u_0))
end function

function TreePolicy(node)
    while not TerminalTest(State(node)) do
        if node not fully expanded then
            return Expand(node)
        else
            node ← BestChild(node, C_puct)
        end if
        return node
    end while
end function

function Expand(node, C_puct)
    a ← Actions(State(node)) − TriedActions(node)
    state ← Result(State(node), a)
    childNode ← Node(state, action = a, parent = node)
    return childNode
end function

function BestChild(node, C_puct)
    return arg max_{c∈Children(node)} Q(c)/N(c) + C_puct sqrt(2ln N(node) / N(c))
end function

function DefaultPolicy(state)
    while not TerminalTest(state) do
        a ← UniformlyRandom(Actions(state))
        state ← Result(state, a)
    end while
    return Reward(state)
end function

function Backup(node, Δ)
    while node do
        N(node) ← N(node) + 1
        Q(node) ← Q(node) + Δ
        Δ ← −Δ
        node ← Parent(node)
    end while
end function
```

4.2 MCTS Parallelization

One of the advantages of Monte Carlo Tree Search is that we can easily convert it to a parallel algorithm. Since the algorithm is composed of four distinct steps, the different parallelization methods can be distinguish by the step being parallelized[3].

- **Leaf Parallelization**: In this method parallel simulations are executed and the results are propagated back to the tree from a single thread. The biggest disadvantage of this method is that since the simulation threads are independent, no information is shared, which can result in wasting computational power for non promising actions.

- **Root Parallelization**: In this method parallel independent trees are built and explored. The results from all threads are merged in order to depict the resulting move. No information is shared between threads and no significant disadvantages exist for this method.
- **Tree Parallelization**: In this method the whole tree is shared which makes the need of mutexes necessary. There are two ways to lock access to the shared tree: Global mutex that locks the whole tree and allows parallel threads only to run simulations from different leaf nodes or Local mutexes that lock nodes of the tree when a certain thread visits them and unlock them on departure. Another technique is adding a virtual loss to the nodes of the tree indicating exploration by another thread. Nodes with virtual loss will be selected for other threads only if their value is better than their siblings. This forces MCTS to keep a good balance between exploration and exploitation share among all running threads.



Chapter 5

Alpha zero

Monte Carlo tree search despite being an algorithm with very promising characteristics has some disadvantages resulting in weak performance. As stated, simulation and selection strategies are both subject to trade-offs, thus making the convergence to strong play slow. Enhancing strategies with domain-specific knowledge can improve the overall performance, but removes the generality of the algorithm. Alpha Zero is an algorithm based on Monte Carlo tree search enhanced with a neural network producing policy and value approximations for game states. The Alpha Zero is using a self-play technique to improve on the estimations over time.

5.1 Neural Networks

5.1.1 Feedforward Neural Networks

Feedforward neural networks form the basis of the artificial neural networks and provide the theory to build more complex machine learning models. In simple terms, they are represented by composition of different functions and are modeled with a directed acyclic graph defining how these functions are composed. The most common pattern of composition is chaining, thus composing the functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ forms $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. The final layer of the network $f^{(3)}$ is called output layer. For supervised learning, the training data provides examples ($y \approx f^{\star}(\boldsymbol{x})$) of the function we aim to approximate, by specifying the desired result of the output layer. Since the training data does not specify the desired result for the other layer, these are called hidden layers.

Given the target function $f^{\star}(\boldsymbol{x})$ our model targets to provide a good estimation for the function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ by adapting the parameters $\boldsymbol{\theta}$. We will use a simple example of a feedforward network containing one hidden layer. Our networks define two functions composed by the chaining rule: $y = f(\boldsymbol{x}; \boldsymbol{\theta}) = f^{(2)}(f^{(1)}(\boldsymbol{x}; \boldsymbol{\theta}))$ where $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b) = \boldsymbol{h}^T\boldsymbol{w} + b$ is a linear model and $\boldsymbol{h}$ is the output of the hidden layer. Letting $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ the complete model can be written as:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$$

In order to describe approximate non linear functions we should use a non-linear function for $f^{(1)}$. The function used for this purpose is an affine transformation followed by a fixed nonlinear function called activation function.

$$\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$$

where $\boldsymbol{W}$ are the **weights** driving the linear transformation and $\boldsymbol{c}$ are the **biases**. There are many functions that provide non-linearities to feedforward networks. Choosing the activation function to be $g(z) = max\{0, z\}$ we can define the complete networks as:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T max\{0, \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c}\} + b$$

This general principle describes how by providing non-linear transformations ϕ we can extend linear models to represent non linear functions $y = \phi(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$. In the case of deep feedforward networks ϕ is defined as the hidden layer and optimization algorithms are used for finding $\boldsymbol{\theta}$ parameter that provide good approximation for our function.

5.1.2 Gradient-Based Optimization

Given a function $y = f(x)$ its derivative $f'(x) = \frac{dy}{dx}$ defines the slope of $f(x)$ at every point x. We can use the derivative to obtain the effect of a small change in the input to the output: $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$. The derivative can be used to minimize the output of a function by guiding changes to input x in order to improve y. Iterative applying small changes in a direction opposite by the one defined from the derivative can reduce $f(x)$. This technique is called gradient descent. Since most learning algorithms attempt to optimize their loss function (ie how closely their approximation is to the training data) gradient descent is the core algorithm used for achieving this.
Points that satisfy $f'(x) = 0$, convey that the derivative contains no information about the direction to apply changes, and are called critical points. These points can be either a local minimum, a local maximum or a saddle points. It is very common in deep learning loss functions to have local minimum points, that are not optimal, therefore making the optimization process difficult. In multidimensional functions we use partial derivatives to measure changes of f in respect to a single variable $\frac{\partial}{\partial x_i} f(\boldsymbol{x})$ and $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ to denote the vector containing all the partial derivatives. Directional derivative in direction $\boldsymbol{u}$ is the slope of the function f in the u direction: $\frac{\partial}{\partial a} f(x + a\boldsymbol{u})$. Minimizing f is about finding the direction where f decreases with a higher slope:

$$\min_{\boldsymbol{u}, \boldsymbol{u}^T \boldsymbol{u} = 1} \boldsymbol{u}^T \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \min_{\boldsymbol{u}, \boldsymbol{u}^T \boldsymbol{u} = 1} \| \boldsymbol{u} \|_2 \| \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \|_2 \, cos\theta$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. For $\| \boldsymbol{u} \|_2 = 1$ and ignoring parts that do not depend on $\boldsymbol{u}$ this simplifies to $min_{\boldsymbol{u}} cos\theta$ which is minimized when $\boldsymbol{u}$ points opposite to the gradient. So decreasing f can be achieved by moving in a direction opposite to the negative gradient:

$$\boldsymbol{x'} = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$$

The parameter determines ϵ the step size and it's known as learning rate.

5.1.3 Back-Propagation

Feedforward neural network's name comes from the fact that information flows through the function being evaluated, from $\boldsymbol{x}$, through the intermediate computations defining f, and finally to the output y. This information flow from the input towards the output of the network is call forward-propagation. The back-propagation algorithm allows information to flow from the loss function backwards through the network in order to compute the gradients in an efficient way.
Back-propagation utilizes the chain rule that defines the rules for computing derivatives of composite functions. For the composite function $z(y) = f(y)$ where $y = g(x)$ the derivative can be computed as:

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

Generalizing for $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$ and $f\mathbb{R}^n \to \mathbb{R}$ we have:

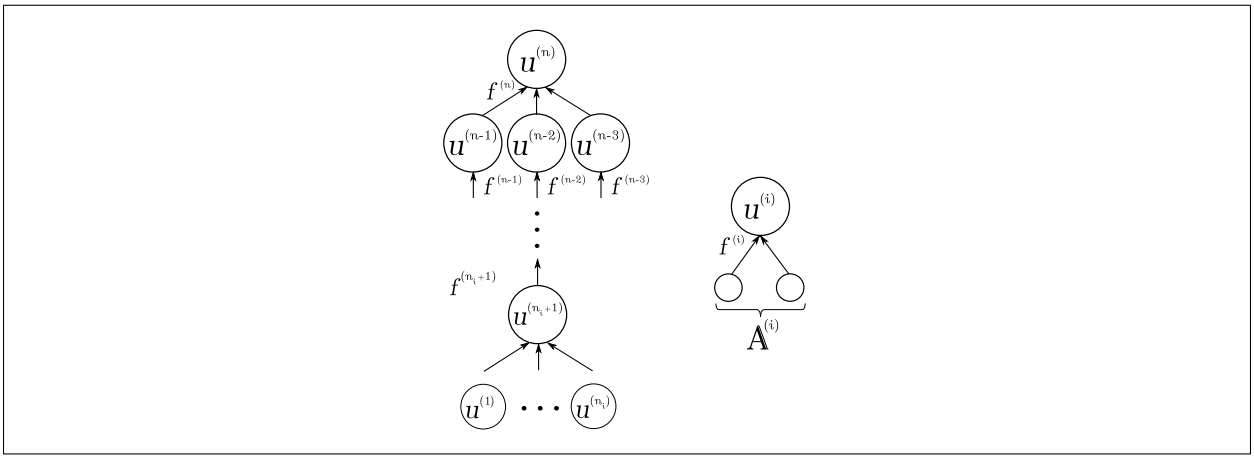
$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x_i}$$

or written using vector notation:

$$\nabla_x = (\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}})^T \nabla_{\boldsymbol{y}} z$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

We will use the terminology of a computational graph to describe back-propagation in feedforward neural networks: each node of the graph represents a variable (scalar, vector, matrix) and an operator that can be applied to one ore more nodes. Operators can be defined returning single output. Directed Edges in the graph from a variable x to a variable y represent that y was computed by applying an operator to variable x. The forward propagation can be described as a computational graph $\mathcal{G}$ that outputs a single scalar $u^{(n)}$. In order to perform back-propagation we construct a computational graph $\mathcal{B}$ that depends on $\mathcal{G}$, where computations are applied in a reverse order.



Algorithm Forward Propagation

for $i = 1, ..., n_i$ **do**
  $u^i \leftarrow x_i$
end for
for $i = n_{i+1}, ..., n$ **do**
  $\mathbb{A} \leftarrow \{u^{(j)} | j \in \textbf{Parent}(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

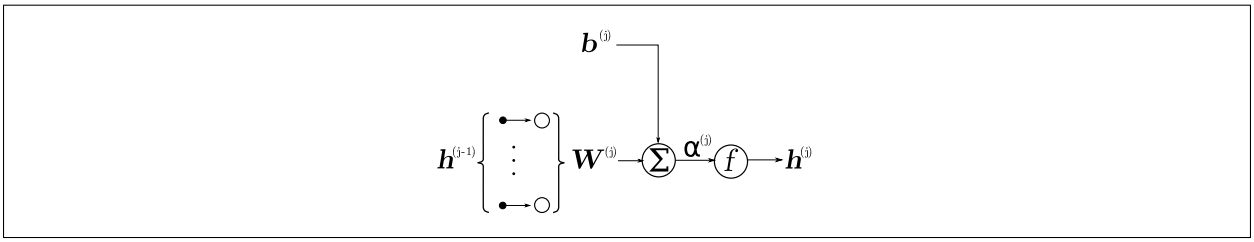
Back-propagation algorithm then computes the derivatives of the output $u^{(n)}$ with respect to the variables in the computational graph $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in 1, 2..., n_i$. The usage of a gradient table $\mathbb{T}[u^{(i)}] = \frac{\partial u(n)}{\partial u(i)}$ storing computed derivatives is needed in order to reduce computations. In more details, chain rule and stored derivates are used to obtain $\frac{\partial u^{(n)}}{u^{(j)}} = \sum_{i \to \{j | j \in \textbf{Parent}(u^{(i)})\}} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ without directly calculating $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. The algorithm visits each edge $u^{(j)} \to u^{(i)}$ once and performs a Jacobian production to obtain the associated partial derivative $\frac{\partial u(i)}{\partial u(j)}$.

Algorithm Back-Propagation

$\mathbb{T}[u^{(n)}] \leftarrow 1$
for $j = n - 1, ..., 1$ **do**
  $T[u^{(j)}] \leftarrow \sum_{i \to \{j | j \in \textbf{Parent}(u^{(i)})\}} T[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$
end for
return $\{T[u^{(i)}] | u = 1, ..., n_i\}$

For mutli-layer fully connected neural networks, the computational graph defines the mapping of parameters, to a loss function $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ as a result of a single example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ being the output on input $\boldsymbol{x}$. Given a network with l layers, let:

- $\boldsymbol{W^{(i)}}$ for $i \in 1, ..., l$ the weight matrices
- $\boldsymbol{b^{(i)}}$ for $i \in 1, ..., l$ the biases parameters



To obtain the total cost J, a regularized $\Omega(\theta)$ may be added where θ contains all the parameters of the model (weights and biases).

Algorithm NN Forward Propagation

$\boldsymbol{h^{(0)}} \leftarrow \boldsymbol{x}$
for $j = 1, ..., l$ **do**
  $\boldsymbol{a}^{(j)} \leftarrow \boldsymbol{b}^{(j)} + \boldsymbol{W}^{(j)}\boldsymbol{h}^{(j-1)}$
  $\boldsymbol{h}^{(j)} \leftarrow f(\boldsymbol{a}^{(j)})$
end for
$\hat{\boldsymbol{y}} \leftarrow h^{(l)}$
$J \leftarrow L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)$

For back-propagation we need to compute the partial derivatives $\frac{\partial J}{\partial \boldsymbol{W}^{(j)}}$ and $\frac{\partial J}{\partial \boldsymbol{b}^{(j)}}$. We introduce an intermediate quantity $\boldsymbol{g}$, that will be computed using the back-propagation, and relate it to the the derivatives.

$$\boldsymbol{g} \equiv \nabla_{\boldsymbol{a}^{(j)}} J = \frac{\partial J}{\partial \boldsymbol{a}^{(j)}} = \frac{\partial L}{\partial \boldsymbol{a}^{(j)}}$$

Algorithm NN Back-Propagation

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $j = l, l-1, ..., 1$ **do**
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(j)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(j)})$
  $\nabla_{\boldsymbol{b}^{(j)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(j)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(j)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(j)}} J\Omega(\theta)$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(j-1)}} J\boldsymbol{W}^{(j)T}\boldsymbol{g}$
end for

Applying the chain rule, we can express the partial derivative of the output activations in terms of the partial derivative of weighted sums (pre-nonlinearity activation):

$$\frac{\partial J}{\partial \boldsymbol{h}^{(j)}} \frac{\partial \boldsymbol{h}^{(j)}}{\partial \boldsymbol{a}^{(j)}} \xrightarrow{\boldsymbol{h}^{(j)} = f(\boldsymbol{a}^{(j)})} \frac{\partial J}{\partial \boldsymbol{h}^{(j)}} f'(\boldsymbol{a}^{(j)}) = \frac{\partial L}{\partial \boldsymbol{h}^{(j)}} f'(\boldsymbol{a}^{(j)})$$

Applying the chain rule, we can relate the gradients on weight and biases:

$$\frac{\partial J}{\partial \boldsymbol{a}^{(j)}} \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{b}^{(j)}} \xrightarrow[J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)]{\boldsymbol{a}^{(j)} = \boldsymbol{b}^{(j)} + \boldsymbol{W}^{(j)}\boldsymbol{h}^{(j-1)}} (\frac{\partial L}{\partial \boldsymbol{a}^{(j)}} + \lambda \frac{\partial \Omega(\theta)}{\partial \boldsymbol{a}^{(j)}}) \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{b}^{(j)}} = \boldsymbol{g} + \lambda \frac{\partial \Omega(\theta)}{\partial \boldsymbol{b}^{(j)}}$$

$$\frac{\partial J}{\partial \boldsymbol{a}^{(j)}} \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{W}^{(j)}} \xrightarrow[J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\theta)]{\boldsymbol{a}^{(j)} = \boldsymbol{W}^{(j)} + \boldsymbol{W}^{(j)}\boldsymbol{h}^{(j-1)}} (\frac{\partial L}{\partial \boldsymbol{a}^{(j)}} + \lambda \frac{\partial \Omega(\theta)}{\partial \boldsymbol{a}^{(j)}}) \frac{\partial \boldsymbol{a}^{(j)}}{\partial \boldsymbol{W}^{(j)}} = \boldsymbol{g}\boldsymbol{h}^{(j-1)} + \lambda \frac{\partial \Omega(\theta)}{\partial \boldsymbol{W}^{(j)}}$$

Finally we express the derivative of the weighted sum in terms of the weighted sum for the previous layer and propagate them to the next hidden layer:

$$\frac{\partial J}{\partial \boldsymbol{a}^{(j)}} = \frac{\partial J}{\partial \boldsymbol{a}^{(j-1)}} \frac{\partial \boldsymbol{a}^{(j-1)}}{\partial \boldsymbol{a}^{(j)}} = \frac{\partial \boldsymbol{a}^{(j-1)}}{\partial \boldsymbol{a}^{(j)}} \boldsymbol{g}$$

and since the weighted sum of a layer can be expressed as:

$$\boldsymbol{a}^{(j)} = \boldsymbol{W}^{(j)} \boldsymbol{h}^{(j-1)} + \boldsymbol{b} = \boldsymbol{W}^{(j)} f(\boldsymbol{a}^{(j-1)}) + \boldsymbol{b}$$

$$\boldsymbol{g} = W^{(j)T} \boldsymbol{g}$$

5.1.4 Convolutional Networks

Convolutional neural networks are a specialized feedforward networks for processing data with a grid-like topology. Their name arise from the fact that convolution is used instead of general matrix multiplication. Since game states can be represented as grids, we can treat them as inputs to a convolutional network in order to derive move policy and value approximations.

Convolution is a mathematical operation of two functions x and w that expresses the amount of overlap of one function as it shifted over another function:

$$s(t) = \int x(a) w(t-a) da = (x * w)(t)$$

In convolutional network teminology the function x is called input, the function w is called kernel and the output is called feature map. For discrete functions the integral is replaced with a summation:

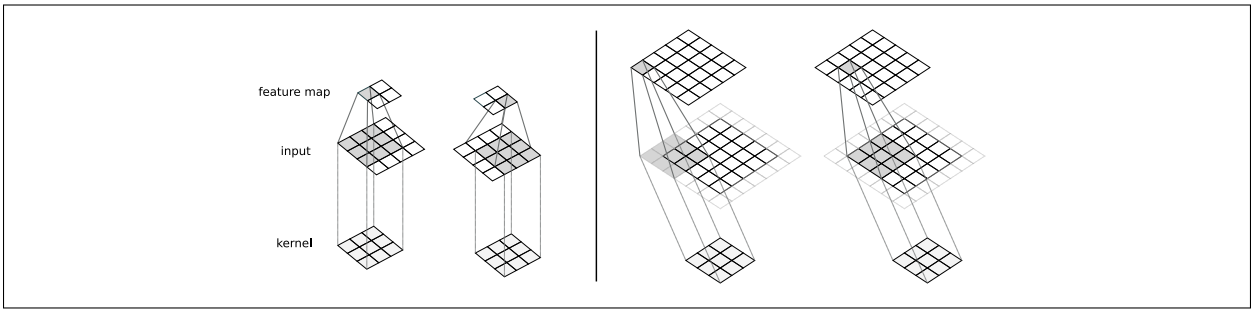
$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a) w(t-a)$$

For two-dimensional input I and kernel K the convolution operation becomes:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n) K(i-m, j-n)$$

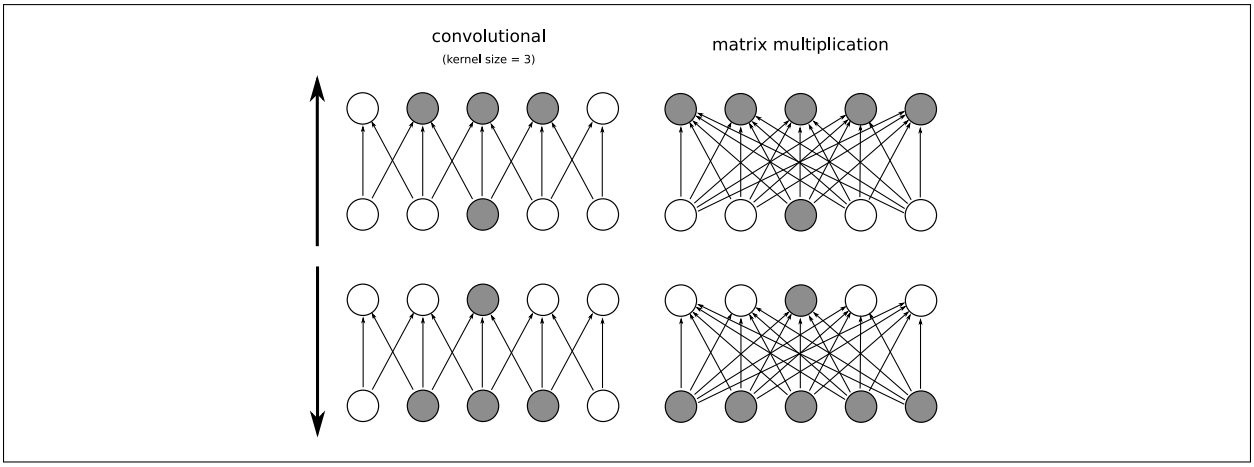
Or since the operation is commutative:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n)$$



Convolution leverages three characteristics that improve a machine learning system:
Using kernels with dimensions smaller than the dimensions of the input, results in **sparse interactions**

between the input and the output units. A tradition network operating with matrix multiplication, uses a different parameter for describing each connection, so every output interacts with every input. For m inputs and n outputs this requires $m \times n$ parameters and $O(m \times n)$ run-time. In contrast, for convolutional networks we can limit the parameters to $k \times n$ with $O(k \times n)$ run-time. This means that each element of the kernel is reused at every convoluted position which we refer to as **parameter sharing**.

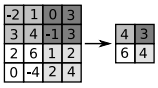


Lastly the parameter sharing in convolution causer layer to have a property, called **equivariance** to translation. This means that translations of input features results in equivalent translation of outputs. This property can benefit for when we know that some function of a small number of neighboring points in the grid is useful when applied to multiple input locations.

Pooling

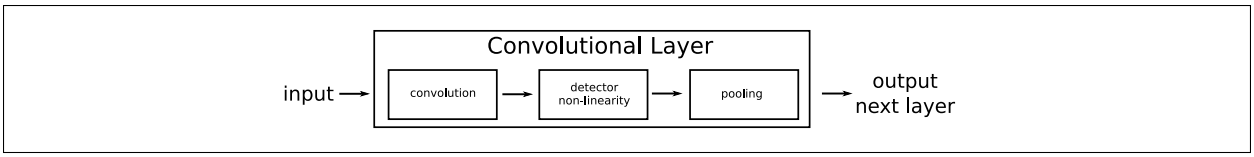
A pooling function is used in order to replace each output of a layer with a statistic summary of the neighboring outputs. This makes the representation approximately invariant to small translations of the input. Invariance to translation as a property can benefit in cases we want to detect a feature's presence instead of a feature's exact location.

The most common pooling strategy is max pooling where the pooling function reports the maximum output within a rectangular neighborhood.



The components of a typical convolutional neural network layer are:

- Convolutional affine transformation.
- Non-linearity activation (most commonly Rectified Linear Unit)
- Pooling (most commonly max-pooling)



5.2 Alpha-Zero Algorithm

Alpha zero is a self-play reinforcement learning algorithm combining Monte Carlo Tree Search and convolutional neural networks.[15]

Monte Carlo tree search selection and simulation processes are replaced by a deep convolutional neural network f_θ parametrised by θ parameters. The input to the network is the 2-dimensional representation of the state s from the prespective of a single player. The network outputs a continuous value of the state $u_\theta \in [-1, 1]$ and a move probability vector $\vec{p}_\theta$ representing the probability of selecting each of the valid moves $p_a = Pr(a|s)$:

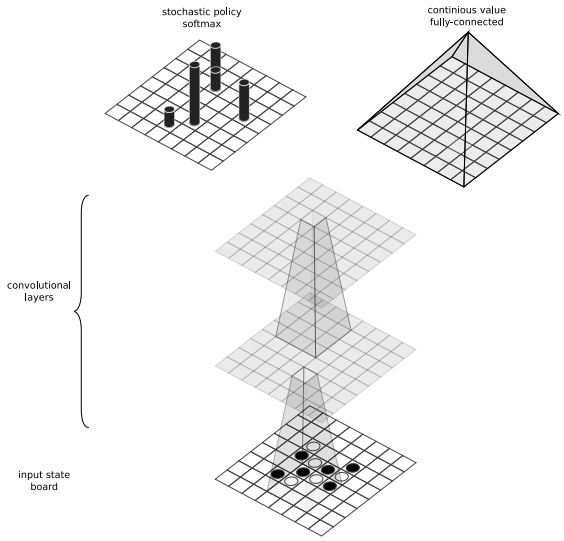
$$(\vec{p}, u) = f_\theta(s)$$

The neural network training data are the result of self-playing games. In particular for each state s_t of the game, an MCTS search α_θ is performed guided by the move probabilities and value given by the best so far neural network f_θ and outputs the probabilities $\vec{\pi}_t$ for each of the applicable actions. The move probabilities $\vec{\pi}$ usually result in stronger moves than the raw probabilities of the network $\vec{p}$. The winner z of each game, along with the improved move probabilities $\vec{\pi}$, are used as training data for the neural network so that $(\vec{p}, u) = f_\theta(s)$ to match the imporved move probabilities and the winner $(\vec{\pi}, z)$. The parameters produced from the training are used in the next iteration of the self-play to strengthen the search even more.

Algorithm ExecuteEpisode

```
function ExecuteEpisode(θ)
    examples ← []
    state ← game.InitialState()
    while do
        for i = 1, .., numberOfSimulations do
            MCTS(state, θ)
        end for
        examples.append(state, π_state,)
        state ← Result(state, action)
        if GameEnded(state) then
            examples ← assignRewards(examples)
            return examples
        end if
    end while
end function
```

The parameters θ of the network are updated in order to minimize the difference of the policy vector $\vec{p}_t$, the search probabilities $\vec{\pi}_t$ and the error between the predicted u_t and z winner of the game. The new parameters are then used for the next self-play iteration of the algorithm.



The outputs of the neural network given a game state are generated by a series of residual layers. The move probabilites are calculated using a stochastic softmax policy that normalizes the sum of the outputs to 1. The value of the state from the prespective of the current player is calculated using a fully connected layer that normalizes the output between -1 and 1.

Monte Carlo Tree Search is similar to the original algorithm with the addition of the outputs of the neural network for each state. In particular, each simulation traverses the game tree by selecting the maximizing Q action and an upper confident bound U dependent on the prior probability P and the total visit count N for this edge. A leaf node s is then expanded using the neural network $P(s,\cdot), V(s) = f_\theta(s)$ and the vector of P values are stored in the outgoing edges of s. Action values Q are updated to track the mean value of all evaluations in the subtree below. The search probabilities $\vec{\pi}$ are returned when the search is completed, and are proportional to $N^{\frac{1}{\tau}}$ when N the total visit count of each edge from the root state and τ a temperature parameter controlling the exploration/exploitation of the search.

Algorithm PolicyIteration

```
function PolicyIteration
    θ ← initializeNeuralNetwork()
    examples ← []
    for i = 1, .., numberOfItems do
        for e = 1, .., numberOfEpisodes do
            episodeResult ← executeEpisode(NeuralNetwork)
            examples.append(episodeResult)
        end for
        θ_new ← trainNeuralNetwork(examples)
        if WinRatio(θ_new, θ) ≥ threshold then
            θ ← θ_new
        end if
    end for
end function
```

The training of the neural network is conducted by a series of self-playing reinforcment algorithm. The network is initialized with random weights θ_0, and at each subsequent iteration i a number of self-play games are executed. At each step t and MCTS search is executed using the weights resulted from the previous iteration $f_{\theta-1}$, and the selected action is sampled from the search probabilities $\pi_t = \alpha_{\theta-1}(s_t)$. The game is played until a terminal state and scored to produce the reward $r_T \in \{-1, +1\}$. The date for each time step t are store as $(s_t = \vec{\pi}_t, z_t)$ where $z_t = \pm r_T$ is the winner from the prespective of the current player.

Algorithm AlphaZero

function AlphaZero($state$)
  $u_0 \leftarrow Node(state)$
  while within budget **do**
    $u_i \leftarrow$ TreePolicy(u_0)
    $\Delta \leftarrow$ DefaultPolicy($state(u_i)$)
    Backup(u_i, Δ)
  end while
  return $action$(BestChild(u_0))
end function

function Expand($node, C_{puct}$)
  $a \leftarrow$ Actions(State($node$)) $-$ TriedActions($node$)
  $state \leftarrow$ Result(State($node$), a)
  $P \leftarrow p_a(state), p_a \in \vec{p}, u = f_\theta(state)$
  $childNode \leftarrow Node(state, action = a, parent = node, P = P)$
  return $childNode$
end function

function BestChild($node, C_{puct}$)
  return $\underset{c \in \text{Children}(node)}{arg\,max} \; Q(c) + C_{puct} P(c) \frac{N(node)}{1+N(c)}$
end function

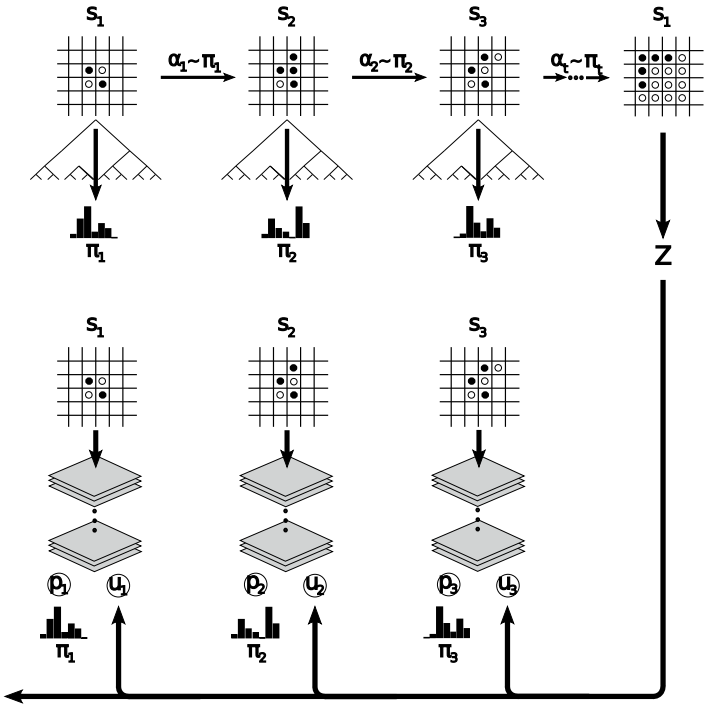
function DefaultPolicy($state$)
  return $u, p_a \in \vec{p}, u = f_\theta(state)$
end function

function Backup($node, \Delta$)
  while $node$ **do**
    $N(node) \leftarrow N(node) + 1$
    $Q(node) \leftarrow \frac{N(node) * Q(node) + \Delta}{N(node)+1}$
    $\Delta \leftarrow -\Delta$
    $node \leftarrow$ Parent($node$)
  end while
end function

The neural network is adjusted by gradient descent on a loss function that sums over the mean-square error and the cross-entropy loss:

$$(\vec{p}, u) = f_\theta(s); l = (z-u)^2 - \vec{\pi}^T log\vec{p} + c \parallel \theta \parallel^2$$

The parameters θ of the network are updated in order to minimize the difference of the policy vector $\vec{p}_t$ and the search probabilities $\vec{\pi}_t$ and the error between the predicted u_t and z winner of the game. The new parameters are then used for the next self-play iteration of the algorithm.



The self-training procedure of the algorithm. During self-play the algorithm generates and stores a tripplete for each move: the game state, the search probabilities as generated from Monte-Carlo Tree Search and the winning player that is assigned to each tripplete when a single game ends. The inputs generated by self-play are sample using the mini-batch technique and fed into the deep neural network that is adjusted based on the loss function. The generated agent is evalutated against the best (at the time) and if the new network's win ration is above 55/100 it is decalred the best.

Chapter 6

Experimental Results

Before expanding into the results, we will be setting the expectations for the experiment.

- Alpha-beta algorithm, enhanced with decent heuristic evaluation functions, has been proven to perform extremely well in a wide variety of games. As we will see later this algorithm is dominating with minimal experimentation, which explains why alpha-beta is the de facto algorithm when creating agents for two player games.
- Monte Carlo Tree Search despite being a vastly different approach its performance is limited, due to the fact that a clean start is made every time the agent searches for the best move. This makes clear that the best behavior for this algorithm is observed when the search tree has a relative small depth. We expect poor performance against other algorithms but strong endgames.
- Alpha-zero algorithm, despite being able to produce the state of art agents for compex games, needs dedicated hardware in order to be able to perform well in a reasonable training time. Since the access to such hardware wasn't possible we don't expect this algorithm to perform as described in the literature, but we hope to see some sophisticated playing.

6.0.1 Baseline Agents

The experiments performed aimed to evaluate these three algorithms against each other. Since it is obvious that the comparison won't be able to give valuable results, we also implemented two baselines that will give back the performance of each algorithm independently. Specifically:

- a random agent, one that will pick it's next move uniformly in random from the available moves.
- a naive greedy agent, one that will pick it's next move in order to maximize a naive metric for each move.

6.0.2 Alpha-Zero models

The models used for the Alpha-Zero algorithm follow the same architecture for both games examined. In the core of the models a configurable number of residual layers. Each layer is composed from:

- a 2D convolutional layer
- a batch normalization layer
- an activation layer using the *relu* activation function
- a 2D convolutional layer

- a batch normalization layer
- a summation layer for combining the results of the input layer with the output layer composed from the previous layers
- an activation layer using the *relu* activation function applied on the result of the previous layer

The residual layers are chained the one after the other. In order to achieve a better performance and give the ability for better generalization (prevent overfitting) a *dropout* layer is added before the results are passed to the policy and the value layers. The architecture of these layers follow the standard pattern, with the value network being a dense layer with a $softmax$ activation and the policy network being a dense layer with a *relu* activation.

6.1 Othello Game

The game of Othello is a two player game, played on an 8x8 grid. The goal of each player is to have the most pieces on the board at the end of the game. A valid move is any piece placed on the grid that will cause one or more opponent pieces to be surrounded vertically, horizontally or diagonally by the player's pieces already on the board. After the move, all opponent pieces surrounded are converted to the player's pieces. The game initial position is with two white and two black pieces placed in the center of the board.

The valid moves for each state can be either found by starting from the current player's pieces and move in any of the 8 posible directions until a blank square is found or the opposite. Depending on the state of the game a different strategy is selected in order to improve the performance of calculating valid moves.

6.1.1 Heuristic evaluation

The game of Othello has been studied in the past and some very strong static evaluation features have been proposed:

- **Coin Parity** Captures the difference of pieces in the board. It is very common for a move to drastically affect this heuristic, so its usage is not suggested especially in early stages of the game.
- **Stability** The most important squares are those on the edges of the board. Stable edge discs are those placed in the edges of the board that can not be flipped. They also provide another great advantage since they can be used as anchors to gain stable internal discs. In order to measure this feature, three classes of pieces are defined: Stable pieces are those that cannot be flipped from the given state, unstable are those that can be flipped in the next move and semi-stable are those that cannot be flipped in the next move but are potentially flippable in the continuation of the game. Each of these classes is assigned with a weight in order to compute the value of this heuristic.
- **Mobility** The greater number of possible moves, the better the ability to exercise more powerful plays and control the proceeding of the game. Mobility is a heuristic that compares the mobility (possible moves) for each player. There exist two flavors of mobility: actual mobility is the possible moves for a given state of the game and potential mobility is the possible moves that might be available in the next few moves.
- **Corners Captured** The corners of the board are by definition stable because they cannot be flipped, but also allows stable expansion around them. The result of a game is highly correlated with the corners captured.

[11][11] All of the above heuristic can be easily calculated by using matrix operations against the board.

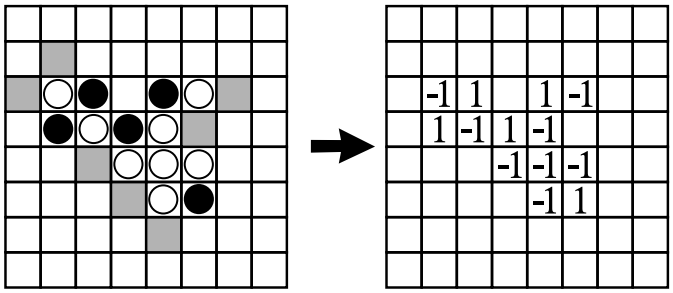
6.1.2 Monte Carlo Tree Search

For the selection step of the algorithm the Upper Confident Bounds method is used.
The move selection is made using the Max method (selecting the child with the highest back-propagated value)

6.1.3 Alpha zero

Games similar to Othello can be easily adapted to the Alpha zero algorithm. Specifically:

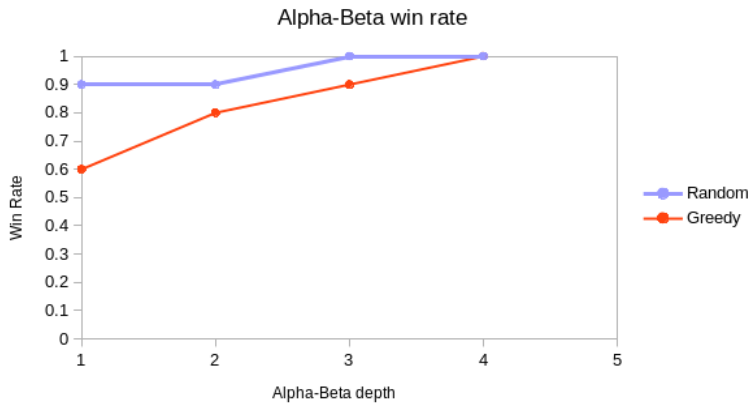
- The input layer for the neural network is 8x8 matrix with zero values in the positions unoccupied, 1 in the positions of the player the algorithm controls and -1 in the positions of the opponent. This representation of the board is intuitive and computationally cheap.
- The board is symmetric under 8 different transformations. This is a great advantage, since instead of a single training point created from each position, we can create 8 training points and in extend train our algorithm faster and more accurate.



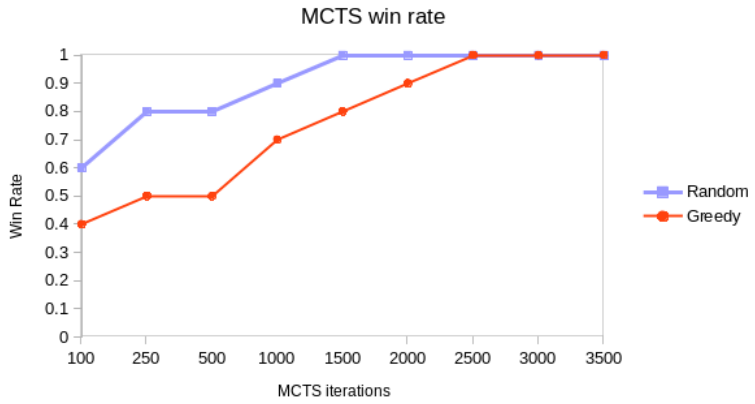
The transformation of the board to the input layer for the CNN used in Alpha-Zero. There are 7 more input layers produced from each state of the Othello game, since the game is invariant to flips and rotations of the board.

6.1.4 Results

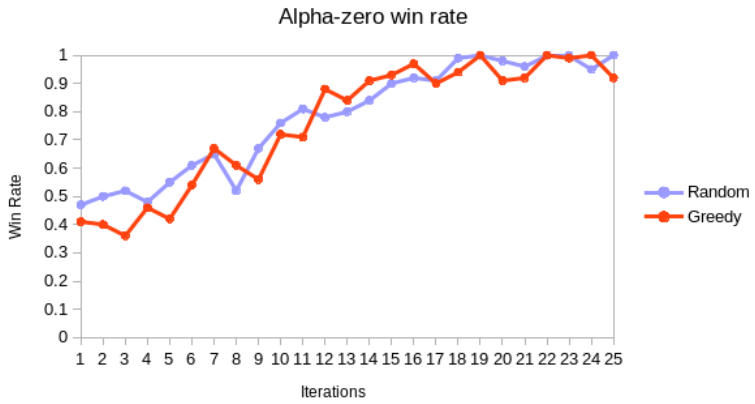
The three algorithms were evaluated under two agents. The random agent performed random moves in each turn and the greedy algorithm performed the move that was resulting in the majority of stones on each turn. Due to the fact that the branching factor of the Othello game is relativelly low, there is a measurable probability of even random moves to produce a strong play. For the greedy strategy, it's obvious that it cannot generate comparable strong plays, but it can directly compete agents that haven't been paremetized or trained in the desired accuracy.



The win rate of the Alpha-Beta algorithm against the baseline algorithms. The algorithm is dominating even for shallow search depth and is able to consistently outperform on bigger search depths.



The win rate of the MCTS algorithm against the baseline algorithms. The algorithm can not perform well on small number of iterations, but it starts dominating on 2000 iterations and above where it can explore a big part of the search tree and select the move with biggest significance level.



The win rate of the Alpha Zero algorithm against the baseline algorithms. The algorith starts to outperform both baselines after 15 iterations of training. Examining the strategies followed from the final agent reveal some very interesting and strong patterns.

6.2 Quoridor Game

The game of Quoridor is a two player game, played on an 9x9 grid. Each player's pawn starts on one side of the board in the center spaced. The goal is to reach the other side of the board, regardless of the position. Each player is equiped with 10 walls that can be placed (fully) on the board, effectively removing two connections between two connected positions.
On a turn, a player may take one action: either move their pawn or place a wall. The valid positions for a pawn is the four positions (up, down, left, right) directly neighboring a position. If a wall is placed between the position and the desired postion this move is invalid. When the two pawns are directly neighboring, instead of the two pawns ending up in the same position, the playing player can move his pawn two positions in the direction of the opposing pawn; in the special case that this position is blocked from a wall the player can choose to place his pawn to any other available neighboring position. Walls can be place anywhere in the board with the only limitation beeing that at least on path towards the goal should exist for both players.

6.2.1 Heuristic evaluation

We propose the following evaluation features:

- **Distance difference** The distance of a pawn towards its goal is the most obvious feature. In order to calculate this distance, we used the Dijkstra[11] algorithm on an undirected graph that represents the board. As starting node the current position of a player is set and as target nodes the target positions.
- **Remaining walls** Walls take an important role in the game since they can be used to block or disrupt the opponent or even push a pawn towards the goal.
- **Alternative path** One of the best tactics in quoridor is to push your opponent on the one side of the board and later block all paths from this side forcing a long path. This makes clear that a player either wants a single path to the goal, since this path cannot be blocked, or an alternative path as short as possible.

6.2.2 Non-Heuristic methods

Since the game of Quoridor has a big branching factor the adoption for non-heuristic methods becomes more expensive. In order to improve the performance of those algorithm the following techiques have been applied:

- A fence cannot potential block all the paths to the goal for any of the player, we use the following strategy to reduce calculations for fence placement: The board is inspected and only fences that intersect with other fences or are positioned in the edges of the board are evaluated for invalid moves. This techique greatly reduces the execution time for finding the applicable moves for each state.
- When all the fences of both playes have been placed, the game is deterministic assuming each player moves using the optimal strategy. We are taking advantage of this by considering a terminal state of the Quoridor game, the state where both players have placed their fences. The winner of the game is calculated by simulating the rest of the game where each player moves towards his goal following the shortest path as calculated using the A-Star[11] algorithm.

6.2.3 Monte Carlo Tree Search

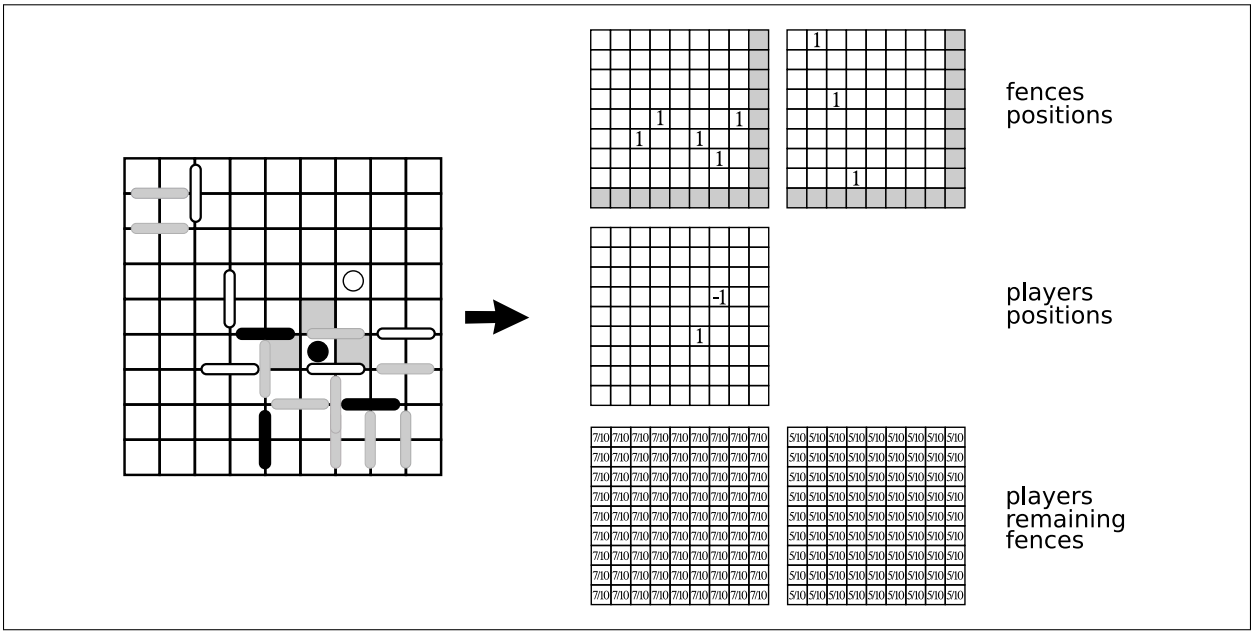
For the selection step of the algorithm the Upper Confident Bounds method is used.
The move selection is made using the Max method (selecting the child with the highest back-propagated value)

6.2.4 Alpha zero

The game state for Quoridor is composed by five 9x9 matrixes. Specifically:

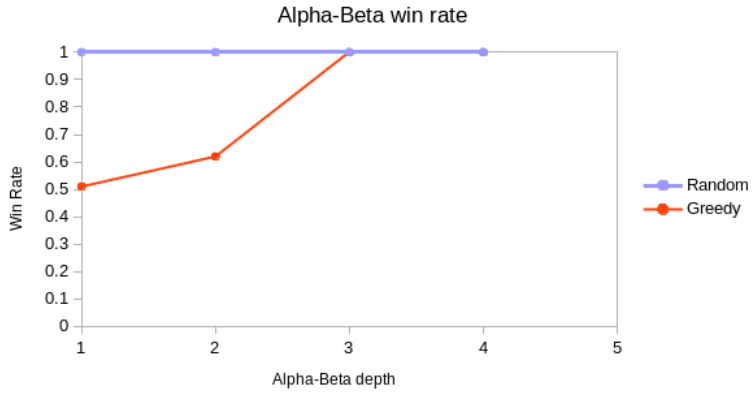
- The first layer points the positions of the players in the board. The current player is marked as 1 in the layer and the opponent player as -1.
- The fence positions are splitted in two layers. The one contains the information for the fences placed horizontally in the board and the other for fences placed vertically. Both for the vertical and the horizontal fences, coordinates that are marked denote a fence being placed underneath and on the right.
- Two layers are dedicated for remaining fences. The remaining fences do not contain positional information, therefore these two layers contain for every point the number of the remaining fences normalized.

This structure may seem complex but it has been chosen so that the fence position not to contain player information.

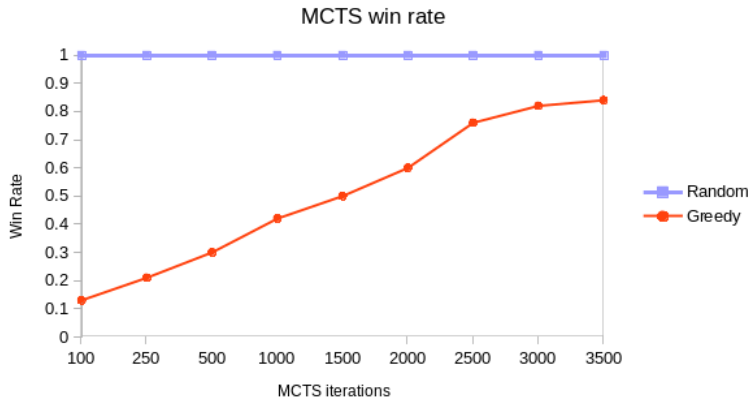


6.2.5 Results

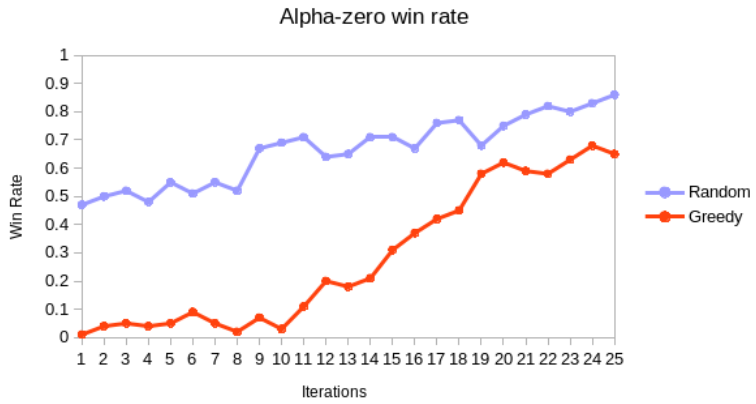
The three algorithms were evaluated under two agents. The random agent performs random moves in each turn. The greedy algorithm performs a pawn move on the shortest path towards the goal, if the fences available are greater or equal to the fences available for the opponent. On the other hand, a fence move blocking the shortest path of the opponent towards the goal is prefered, if the fences available are less than the fences available for the opponent. Considering that the branching factor of the Quoridor game is high and the game can progress infinitely without a winner, a random agent cannot compete even a naive agent.



The win rate of the Alpha-Beta algorithm against the baseline algorithms. The algorithm is dominating even for shallow search depth and is able to consistently outperform on bigger search depths.



The win rate of the MCTS algorithm against the baseline algorithms. The algorithm cannot perform well on small number of iterations but it starts dominating on 3500 iterations and above. The width of the game tree as well as the cyclic behavior of the game (moves can repeat) don't benefit the MCTS algorithm and the produced agents are evolved with a slow rate. When no fences are left the accuracy of the predictions is very accurate.



The win rate of the Alpha Zero algorithm against the baseline algorithms. The algorith has an average performance against both the greedy and the random agents. Even after 25 iterations the win rate starts to lean towards the Alpha-Zero agent. It is clear that the algorithm is improving on every iteration, and consequently can dominate the other agents if trained for long enough.

Chapter 7

Conclusion and Future Work

Two-player games provide an excellent setup for experimenting and improving Artificial Intelligence methods. In this thesis we evaluated 3 algorithms that operate on this base, with very different characteristics. The usage of domain dependent methods is known to suffer generating good results for complex games, where a static evaluation is difficult to be constructed. The foundation for generating agents using domain independent methods has been given with the Monte Carlo Tree Search algorithm. The drawback of this algoritmh is that is online (in the sense that no pre-computation has been made) and that any results are not available for next runs. These drawbacks have been overtaken by the Alpha-Zero algorithm which fundamentally performs the tree search in an offline step and uses the results to train a neural network. The neural network itself can be thought as an optimized and compressed way to store the knowledge gained from previous games. This algorithm itself has been vastly improved in respect to the performance of the resulting agents, but lacks in the time needed for the offline training step. It may seem tempting to enhance these algorithms with domain specific knowledge in order to explore only the relevant part of the trees, but in contrast, this will probably lead the agents to perform the same as the ones produced from domain specific methods.

In more detail, alpha-beta algorithm has achieved good results for both games considering that good evaluation functions were used to infer the static value of a state. For MCTS the results were impressive given the fact that the algorithm did not used any available knowledge, but its weakness has shown when performing of large game trees. Finally, the alpha-zero algorithm rightly has the state-of art position in the field. Despite the fact that is unable to provide outstanding results, it is able to improve the resulting agent on every run.

Since the area of traditional algorithms has been reasearched in depth, we propose further work to target agents that leverage the advantages of neural networks:

- Experiment with different state representation for the Quoridor game for the Alpha-Zero neural network
- Determine the effect on the performance and the training time for Alpha-Zero based on the number of the residual layers of the model.
- Explore parallelization methods for the training phase of Alpha-Zero.

Bibliography

[1] B. Abramson. "Expected-outcome: a general model of static evaluation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (Feb. 1990).

[2] Bernd Brugmann. *Monte Carlo Go*. 1993.

[3] Tristan Cazenave and Nicolas Jouandeau. "On the parallelization of UCT". In: (Jan. 2007).

[4] Guillaume Chaslot et al. *Monte-Carlo Strategies for Computer Go*. 2006.

[5] Remi Coulom. *Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search*. 2006.

[6] Levente Kocsis and Csaba Szepesvari. "Bandit Based Monte-Carlo Planning". In: ECML 06 (2006), pp. 282–293.

[7] Richard E. Korf. "Depth-first Iterative-deepening: An Optimal Admissible Tree Search". In: *Artif. Intell.* 27.1 (Sept. 1985).

[8] Paul Fischer Peter Auer Nicolo Cesa-Bianchi. "Finite-time analysis of the multiarmed bandit problem". In: (2002).

[9] A. Reinefeld and T. A. Marsland. "Enhanced iterative-deepening search". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16.7 (July 1994), pp. 701–710. ISSN: 0162-8828. DOI: 10.1109/34.297950.

[10] Herbert Robbins. "Some aspects of the sequential design of experiments". In: 58 (1952).

[11] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach (3rd edition)*. 2009.

[12] J. Gillogty S. Fuller J. Gaschnig. "Analysis of the alpha-beta pruning algorithm". In: *Carnegie-Mellon University, Computer Science Department Report* (1973).

[13] J. Schaeffer. "The History Heuristic and Alpha-Beta Search Enhancements in Practice". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 11.11 (Nov. 1989), pp. 1203–1212. ISSN: 0162-8828. DOI: 10.1109/34.42858. URL: https://doi.org/10.1109/34.42858.

[14] Claude E. Shannon. "XXII. Programming a computer for playing chess". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314 (1950), pp. 256–275.

[15] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: 10.1126/science.aar6404. eprint: https://science.sciencemag.org/content/362/6419/1140.full.pdf. URL: https://science.sciencemag.org/content/362/6419/1140.

[16] James R. Slagle and John E. Dixon. "Experiments With Some Programs That Search Game Trees". In: *J. ACM* (Apr. 1969).

[17] Albert L. Zobrist. "A New Hashing Method With Application for Game Playing". In: (1970).