Artificial Intelligence - EDAP01
Assignment 1 - Adversarial Search

Connect Four Game
Minimax Algorithm - Alpha-Beta Pruning

Hicham Mohamad (hi8826mo-s)

24 February 2021

1 Introduction

In this assignment we are going to work on an **artificial intelligence** project that involves implementing the well-known adversarial search algorithm **Minimax** with **Alpha-Beta pruning** algorithm in Python. These algorithms are useful tools to optimize **decision making** for an AI agent. At the end, we will implement a program/agent playing and winning the game Connect Four. At the end, in section 4 we study and summarize the paper about AlphaGo of David Silver et al., which introduce a new approach of Monte Carlo Tree Search algorithm.

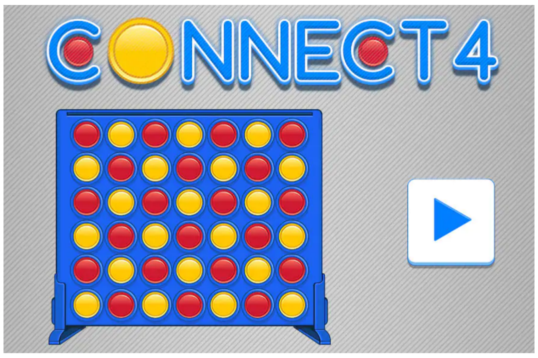


Figure 1: Connect Four game.

1.1 User guide and feedback

The implemented program can be found in the jupyter notebook. Fortunately the code does work nicely now. The report is revised and updated.

The important issue/bug with my Alpha-Beta, when I showed it to you last time, was that I was using `copy()` of the BOARD and in the same time using the function `env.available_moves()` so in this way I was playing two parallel games one by simulating a copy and another is the actual evn for having the available moves. So the solution was to copy the env and not the board !!!

The peer review comments on the solution of the student **Timothy Burfield (spa15tbu)** are found in section 3.

1.2 Objectives

The main task in this assignment is to implement **agent/program** playing a full Connect Four game against the **course game server**. Because we are dealing with **competitive environment**, we need to implement the classical **mini-max adversarial search** algorithm with the **alpha-beta pruning** introduced. The objectives of this assignment consist of the following:

1. Play and not loose against the course game server. Connect Four is a **solved game** and one may always at least get a draw.
2. Not loose consistently in a row of 20 games. The server will store id and the results of the implemented games in order to verify this.
3. Implement at least the classical mini-max adversarial search algorithm, with the alpha-beta pruning introduced for the **best move**.
4. The program should be fast, so a move needs to be decided within max 5 seconds. For that one might need to cut the search tree and introduce some evaluation function.
5. Peer review: comment on some other student's solution. Then in the report we need to take up peer's solution and compare it.
6. Read and summarize a paper of David Silver et al. about AlphaGo: *Mastering the game of Go with deep neural networks and tree search.*

1.3 Playing against the Server

To make a move against the server do a **https request** using the following scheme, which is included in the function `call_server(move)`:

```
res = requests.post(SERVER_ADRESS + "move",
                    data={
                        "stil_id": STIL_ID,
                        "move": move,
                        "api_key": API_KEY,
                    })
```

`SERVER_ADRESS`: should be https://vilde.cs.lth.se/edap01-4inarow/

`STIL_ID`: should be a list with strings containing the stil id:s of the students in the group

`move`: is your move (each new game is starting by sending the move -1)

`API_KEY`: is a key required to get access to the server, this year that key is '`nyckel`'

The standard move notation is used, both for input and output, i.e. X where X is one of **a-g columns**. In this game we don't need to use Y which is one of 1-6 rows. The response you get from the server is:

- **status**: boolean for whether bad stuff happened
- **msg**: the return message describing what happened serverside
- **botmove**: the bot's move
- **result**: if the game ended what points you got, 0 if game is not over
- **state**: a list of lists describing the state with 1:s as your pieces and -1:s as the bots pieces. When receiving the state from the server, one can decide how to go with it, there are several ways: one can use the "env" object and `reset()` it, use the "`step()`" function, or write its own class for it.

1.4 Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball: `https://gym.openai.com/`

Gym provide the **environment**; We need to provide the algorithm. One can write its own agent using the existing numerical computation library, such as **TensorFlow** or **Theano**.

gym and pygame come with the template and are included in the zip-file. It is possible to use gym, it's free code available under MIT license.

1.5 Connect Four game

As mentioned in [5], Connect-Four is a game in the Tic-Tac-Toe family, i.e. a two-player connection board game. the object is to get **four stones** in a row horizontally, vertically or diagonally. There is a **gravity rule**: *you can play only in the bottom-most unoccupied cell in a column.* This means **Zugzwang** can arise, i.e. a situation in which the obligation to make a move is a serious disadvantage. The characteristics of this game can be summarized in the following:

- Players choose a color and then take turns dropping colored discs into a **seven-column** × **six-row** vertically suspended grid.
- The pieces fall straight down, occupying **the lowest** available space within the column.
- The objective of the game is to be the first to form a horizontal, vertical, or diagonal line of four of one's own discs.
- Connect Four is a **solved game**. The first player can always win by playing the right moves.
- Connect Four is a two-player game with **perfect information** for both sides.
- Connect Four also belongs to the classification of an adversarial, **zero-sum game**, since a player's advantage is an opponent's disadvantage.
- The artificial intelligence algorithms able to strongly solve Connect Four are **minimax** or negamax, with optimizations that include alpha-beta pruning, move ordering, and transposition tables.
- With perfect play, the first player can force a win, on or before the 41st move by starting in the **middle column**.
- by starting with the **four outer columns**, the first player allows the second player to force a win.

2 Program implementation

The program for running the game Connect4 is implemented in Python. In the following, we describe the principle methods in the class `ConnectFourEnv` used in the implementation of the algorithms and the provided skeleton. It also includes an inner class `StepResult(NamedTuple)`.

1. `__init__(self, board_shape=(6, 7), window_width=512, window_height=512)` constructor to initializes the board game env.
2. `change_player(self)` change the player to opponent.
3. `step(self, action: int)` return the reward as it determines if the game was won or not. env is constantly updated every time we do env.step().
4. `board(self)` returns a copy of the board state. The only one argument "self", in python, is often used to say that it is a function of the class.
5. `reset()` return the updated game board state
6. `is_valid_action()` verify a legal move, or admit there is no such move by returning False.
7. `available_moves()` get a list of all the valid moves.
8. `is_win_state()` return if this is a win state.
9. `opponents_move(env)` returns a move 0-6 or -1 if it could not make a move.
10. `student_move(env)` should return a move from 0-6 which correspond to a-g columns.
11. `play_game(vs_server = False)` main function for playing Connect4.

2.1 Mini-max adversarial search algorithlm

As shown in Figure 2, a game can be represented by a **game tree** where the nodes are game **states** and the edges are **moves**. Connect4 game can be formally defined as a kind of **search problem**. As almost all game playing programs, our Connect4 player uses a **Minimax search with alpha-beta pruning**. The core idea behind this algorithm is *to choose a move to position with highest minimax value, i.e. best achievable payoff against best player.*

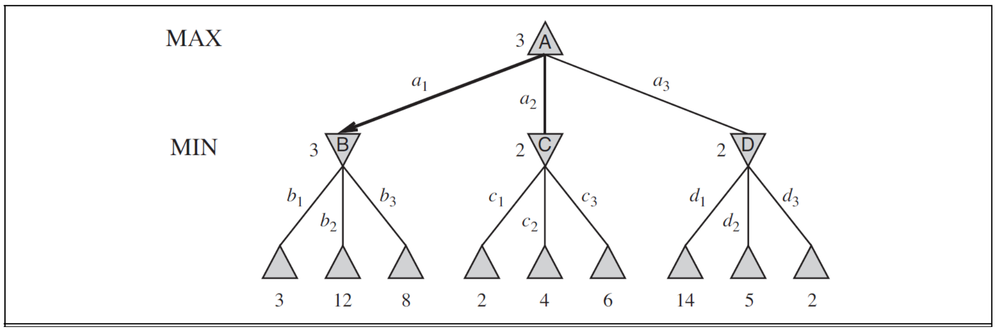
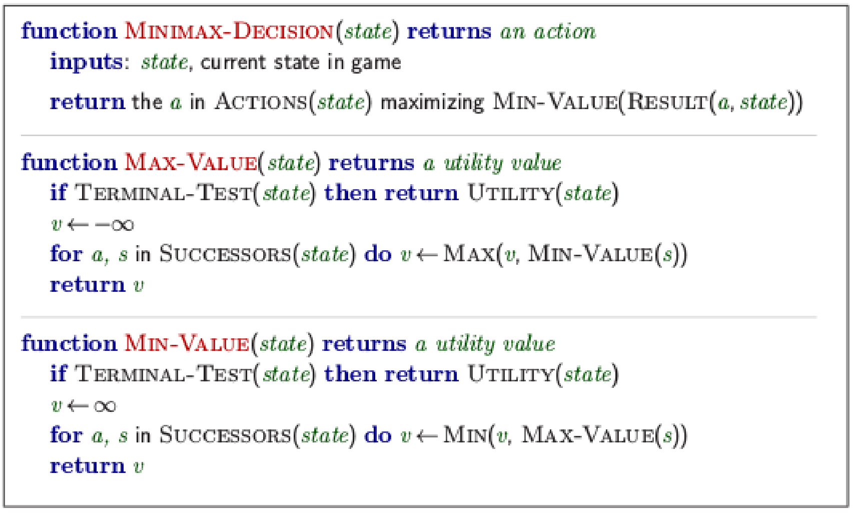


*Figure 2: Schematic of the two-ply game tree. The △ nodes are "MAX nodes", in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes". The terminal nodes show the **utility values** for MAX; the other nodes are labeled with their **minimax values**. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value (3), and MIM's best reply is b_1, because it leads to the state with the lowest minimax value (3).*

The minimax algorithm, as shown in Figure 3, computes the **minimax decision** from the current state. Then it uses a simple **recursive computation** of the minimax values of each successor state, directly implementing the defining equations.



```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game
   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
   return v
```

*Figure 3: **An algorithm for calculating minimax decisions.** It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize the utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state (Aima book page 166).*

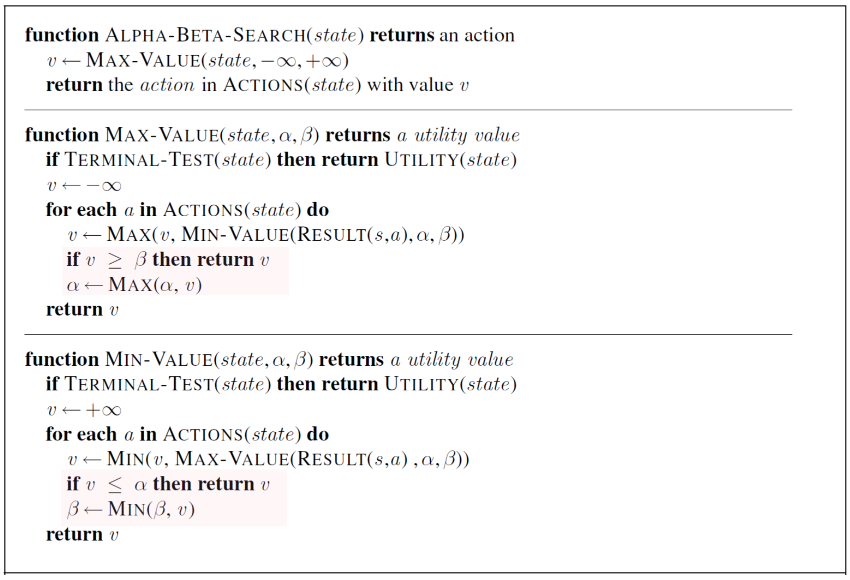
The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. But according to [2], it is possible to compute the correct minimax decision without looking at every node in the game tree using the idea of **pruning**. This technique allows us to ignore and eliminate branches of the search tree that make no difference to the final decision.

2.2 Alpha-beta pruning

Alpha-beta pruning is considered as a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree and it achieves higher efficiency by eliminating subtrees that are irrelevant. The effectiveness of alpha-beta pruning is highly **dependent on the order** in which the states are examined.

Algorithm It maintains two parameters, α and β, which represent respectively the **minimum score** that the maximizing player MAX is assured of and the **maximum score** that the minimizing player MIN is assured of. In other words, these α and β are the values of the **best choice** we have found so far for MAX or MIN respectively. *Initially, α is negative infinity and β is positive infinity, i.e. both players start with their worst possible score.* Then alpha-beta search updates the values of α and β as it goes along and prunes the remaining branches at a node as soon as the value of the current node is known to be worse than the current α or β value.

As shown in Figure 4, whenever the maximum score β that the minimizing player MIN (i.e. the "beta" player) is assured of becomes less than the minimum score α that the maximizing player MAX (i.e., the "alpha" player) is assured of (i.e. $\beta \leq \alpha$), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.



```
function ALPHA-BETA-SEARCH($state$) returns an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value $v$

function MAX-VALUE($state, \alpha, \beta$) returns $a$ $utility$ $value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$,$a$), $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE($state, \alpha, \beta$) returns $a$ $utility$ $value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$,$a$) , $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Figure 4: The alpha-beta search algorithm. We notice that these routines are the same as the MINIMAX functions in Figure 3, except for the two lines in each of MIN_VALUE and MAX_VALUE that maintain α and β. (Aima book page 170)

In the implementation shown in Listing 1, **Mini-max adversial search** algorithm with Alpha-beta pruning consists of a class and includes three functions according to the algorithm illustrated in Figure 4. The first one is `minimax()` which is the general function acting to get the min or the max value based on the depth it currently explores. Function `max_play()` is aimed to maximize the score of the player, whereas the function `min_play()` is aimed to minimize the score of opponent.

2.3 Evaluation function and cutoff test

The minimax algorithm generates the entire game **search space**, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time.

Claude Shannon's paper *Programming a Computer for Playing Chess* (1950) proposed instead that programs should cut off the search earlier and apply a **heuristic evaluation function** to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha–beta in two ways:

- replace the utility function by a **heuristic evaluation function** EVAL, which estimates the position's utility,
- replace the terminal test by a **cutoff test** that decides when to apply EVAL.

An evaluation function returns an estimate of the **expected utility** of the game from a given position. The performance of a game-playing program depends strongly on the quality of its evaluation function [2]. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost.

Heuristic evaluation and Move ordering In this assignment, we provide the adversarial search algorithm with an evaluation function for guiding the search, and limit its search depth. The implemented heuristic evaluation function consists of scoring the choice of position/column in the game board by giving highest score for move starting in the **middle column** 3, and lowest score for moves starting in the **four outer columns** 0, 1, 5 and 6. An average score, lower than that given to the middle column is estimated for columns 2 and 4. These ways of judging the value of a position return to the experience and conclusions obtained in [5] and [6]. They mentioned that the first player can force a win by starting in the middle column. Whereas by starting with the four outer columns, the first player allows the second player to force a win. The **order** in which moves are considered has also been optimized to improve the performance of Alpha Beta Pruning. It starts in the middle and alternating one column left and right. Additional different scores come from evaluating the streaks of different length from 2 to 4 played discs.

Future improvement - Iterative deepning In future work, the evaluation function can be improved depending on **time flow**. The whole point is to search faster without long computation. Thus, it is possible to use the **time complexity** in microsecond as a cutoff test by using `datetime` module in Python and set the max time to some suitable value. In this way, iterative deepening with move ordering can be applied, such that when time runs out the program returns the move selected by the deepest completed search.

Listing 1: Python implementation of the algorithm Minimax with Alpha-beta pruning

```
import copy
#from datetime import datetime

MAX_TIME = 100000

class Optimal_Minimax:

    #def __init__(self, currBoard):
    def __init__(self, env):

        #self.time = datetime.now().microsecond

        #col = self.minimax(copy.deepcopy(currBoard), -1000000, +1000000, 4)  # XXXX
        new_env = copy.deepcopy(env)
        col = self.minimax(new_env, -1000000, +1000000, 5)  # XXXX
        self.action = col

    # mini-max adversial search algorithm is divided in three functions:
    # the first one is minimax which is the general function acting to
    # get the min or the max value based on the depth it currently explors.
    # return: an action
    #def minimax(self, board, alfa, beta, depth):
    def minimax(self, environ, alfa, beta, depth):
        values_dict = []
        bestValue = -1000000
        #bestMove = -1

        legalMoves = environ.available_moves() # XXXXX free columns
        orderedMoves = sorted(legalMoves, key=lambda x: abs(3-x))

        for move in orderedMoves:
            # compare the spent time with the the cut off max time
            #if datetime.now().microsecond - self.time >= MAX_TIME:
            #    return bestMove

            #next_board = copy.copy(board) # XXXXX
            new_env = copy.deepcopy(environ)

            #next_board, reward, done, _ = env.step(move)
```

```python
            next_board, reward, done, _ = new_env.step(move)

            #minValue = self.min_play(next_board, alfa, beta, depth-1)
            minValue = self.min_play(new_env, alfa, beta, depth-1)
            values_dict.append((max(bestValue, minValue), move))
            #bestValue = max(bestValue, minValue)
            #if minValue > bestValue:
            #    bestMove = move
            #    bestValue = minValue

            #next_board = env.reset(next_board) # XXXXX

        bestMove = max(values_dict, key=lambda x: x[0])[1]

        return bestMove

    # function max_play is aimed to maximize the score of player
    # it only returns the value.
    #def max_play(self, board, alfa, beta, depth):
    def max_play(self, environ, alfa, beta, depth):
        state = environ.board
        legalMoves = environ.available_moves() # XXXX

        # if TERMINAL-TEST(state) then return UTILITY(ATATE)
        if (environ.is_win_state()):
            #return countscores(state) + depth # XXXXX
            return streaks_eval(state) + depth # XXXXX
        elif (depth == 0) or (len(legalMoves) == 0):
            #return countscores(state)
            return streaks_eval(state) # XXXXX

        # the worst possible score: v <-- -infinity
        bestValue = -1000000

        #for move in legalMoves:
        # ordered ACTIONS
        for a in [3, 2, 4, 1, 5, 0, 6]:
            if a in legalMoves:
                # compare the spent time with the the cut off max time
                #if datetime.now().microsecond - self.time >= MAX_TIME:
                #    return bestValue

                #next_board = copy.copy(board) # XXXX
                new_env = copy.deepcopy(environ) # XXXX

                next_board, reward, done, _ = new_env.step(a)

                #minValue = self.min_play(next_board, alfa, beta, depth-1)
                minValue = self.min_play(new_env, alfa, beta, depth-1)
                bestValue = max(bestValue, minValue)

                #next_board = env.reset(next_board) # XXXXX

                if bestValue >= beta:
                    return bestValue
                alfa = max(alfa, bestValue)

        return bestValue

    # function min_play is aimed to minimize the score of opponent.
    # it only returns the value.
    # the same as Max_play but with roles of alpha, beta reversed
    #def min_play(self, board, alfa, beta, depth):
    def min_play(self, environ, alfa, beta, depth):
        state = environ.board
        legalMoves = environ.available_moves() # XXXX

        if (env.is_win_state()):
            #return countscores(state) + depth # XXXXX
            return streaks_eval(state) + depth # XXXXX
        if (depth == 0) or (len(legalMoves) == 0):
            #return countscores(state)
            return streaks_eval(state) # XXXXX

        # initially beta is positive infinity (high)
        # the worst possible score
        bestValue = 1000000

        #env.change_player() # change to opponent

        # for each a in ACTIONS(state) do
        #for move in legalMoves:
        for a in [3, 2, 4, 1, 5, 0, 6]:
            if a in legalMoves:
                # compare the spent time with the the cut off max time
                #if datetime.now().microsecond - self.time >= MAX_TIME:
                #    return bestValue

                #Nex_board = copy.copy(board) # XXXX
                new_env = copy.deepcopy(environ)

                new_env.change_player() # change to opponent

                next_board, reward, done, _ = new_env.step(a)

                new_env.change_player() # change back to student before
                    returning

                #maxValue = self.max_play(next_board, alfa, beta, depth-1)
                maxValue = self.max_play(new_env, alfa, beta, depth-1)
                bestValue = min(bestValue, maxValue)

                #next_board = env.reset(next_board) # XXXXX


                if bestValue <= alfa:
                    return bestValue
                beta = min(beta, bestValue)

        return bestValue
```

3 Peer Review

In this peer review task, I comment on the solution of **Timothy Burfield (spa15tbu)**. After showing and discuss with him the two solutions, I could conclude the following questions about Minimax code and the evaluation function:

1. The implementation in Python is working nice.
2. We went through the code together and discuss the content pointing on Minimax and Alpha-beta

algorithms and he showed that he could win against the server.

3. It is sufficiently efficient.
4. He follows a similar strategy according to the theory in the lectures and Aima book.
5. It is based on Alfa-beta pruning of a Minimax tree.
6. Heuristic evaluation of nodes is included in the implementation.
7. It consists on scoring the streak of positions in the game board.
8. It is good enough to achieve the objectives of the assignment.
9. His implementation can be improved by adding a cutoff test such that a move is selected within the allocated time.

4 Reading

In this section, we need to read and summarize the paper of David Silver et al. about the game AlphaGo: *Mastering the game of Go with deep neural networks and tree search*.

This paper presents two new approaches to computer game AlphaGo that has been considered as the most challenging for artificial intelligence due to its enormous **search space** and the difficulty of **evaluating** board positions and moves. Using the following components integrated together, their program could achieve a 99.8% winning rate against other Go programs:

- Two trained deep neural networks: **value networks** to evaluate board positions and **policy networks** to select moves.
- New **search algorithm** that combines Monte Carlo simulation with value and policy networks.

In this way, AlphaGo uses Monte Carlo Tree Search (MCTS) algorithm to find its best moves based on knowledge acquired by the new developed move selection and position evaluation functions, using deep neural networks that are trained by a combination of supervised and reinforcement learning.

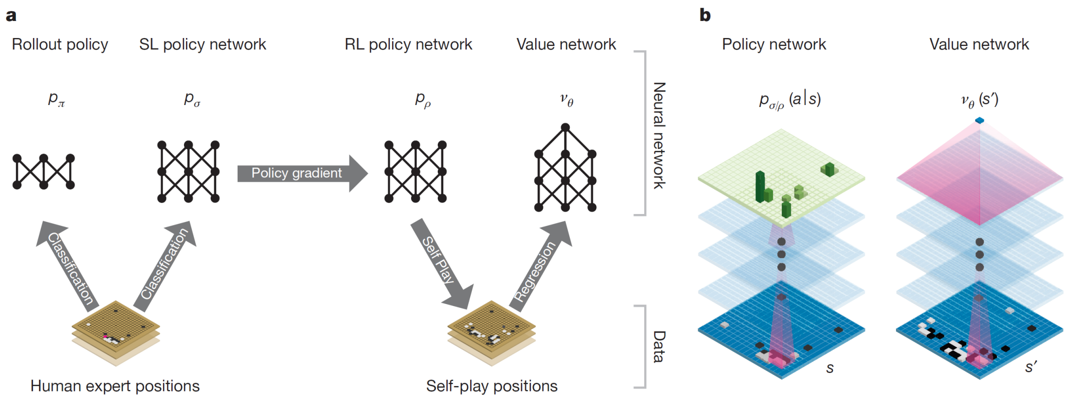


Figure 5: Neural network training pipeline and architecture. ***a)*** *A fast rollout policy p_π and* ***supervised learning*** *(SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A* ***reinforcement learning*** *(RL) policy network p_ρ is initialized to the SL policy network, and is then improved by* ***policy gradient learning*** *to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by* ***regression*** *to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set.* ***b)*** *Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many* ***convolutional layers*** *with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a* probability map *over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.*

4.1 Neural network training pipeline and architecture

By passing the board position as a 19×19 image, they use **convolutional neural layers** to construct a representation of the position. These neural networks are used to reduce the effective depth and breadth of the search tree: **evaluating positions** using a value network, and **sampling actions** using a policy network. As illustrated in Figure 5, the training pipeline consists of several stages and can be characterized by the following points:

- Using human expert moves in a dataset of positions, **supervised learning** (SL) policy network p_σ is trained to provide efficient learning updates and high quality gradients.
- With the same dataset they also train a fast **rollout** policy p_π that can rapidly sample actions during rollouts.
- Next a **reinforcement learning** (RL) policy network p_ρ is trained to improve the SL policy network by optimizing the final outcome of games of self play using **policy gradient learning**. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy.
- With a **self-play dataset** generated by playing games of self-play with the RL policy network, a value network v_θ is trained by **regression** to predict the winner of games played by the RL policy network against itself.

4.2 Tree search algorithm

The program AlphaGo efficiently combines the policy and value networks with MCTS that selects actions by lookahead search, using **Monte Carlo rollouts** to estimate the value of each state in a search tree. The policies are used to narrow the search to a beam of high-probability actions, and to sample actions during rollouts.

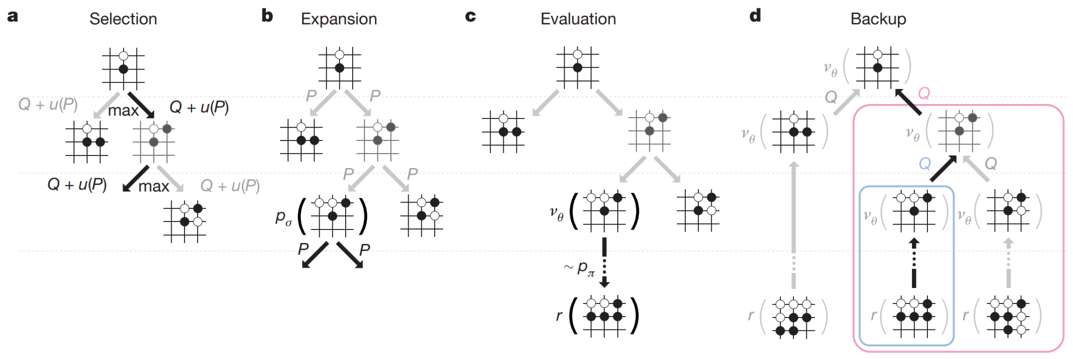


Figure 6: Monte Carlo tree search in AlphaGo. ***a)*** *Each simulation traverses the tree by selecting the edge with* ***maximum action value*** *Q, plus a* ***bonus*** *u(P) that depends on a stored* ***prior probability*** *P for that edge.* ***b)*** *The leaf node may be expanded; the new node is processed once by the* ***policy network*** *p_σ and the output probabilities are stored as prior probabilities P for each action.* ***c)*** *At the end of a simulation, the leaf node is evaluated in two ways: using the* ***value network*** *v_θ; and by running a rollout to the end of the game with the fast* ***rollout policy*** *p_π, then computing the* ***winner*** *with function r.* ***d)*** *Action values Q are updated to track the* ***mean value*** *of all evaluations r() and $v_\theta()$ in the subtree below that action.*

We can summarize the main charateristics of this MCTS algorithm shown in Figure 6 by the following:

- Each edge (s, a) of the search tree stores an action value $Q(s, a)$, visit count $N(s, a)$, and prior probability $P(s, a)$.
- The tree is traversed by simulation, that is, descending the tree in complete games without backup, starting from the root state.
- At each time step t of each simulation, an action a_t is selected from state s_t

$$a_t = \arg\max_a (Q(s_t, a) + u(s_t, a)) \tag{1}$$

where $u(a, s)$ is a bonus that that is proportional to the prior probability but decays with repeated visits to encourage exploration.

- At the end of simulation, the action values and visit counts of all traversed edges are updated. Each edge accumulates the visit count and **mean evaluation** of all simulations passing through that edge.
- Once the search is complete, the algorithm chooses the **most visited move** from the root position.
- Evaluating policy and value networks requires several orders of magnitude more computation than traditional search heuristics.
- To efficiently combine MCTS with deep neural networks, AlphaGo uses an asynchronous **multi-threaded search** that executes simulations on CPUs, and computes policy and value networks in parallel on GPUs.

4.3 Comparison

In this assignment, by using Alpha-beta algorithm the depth of the search may be reduced by **position evaluation**, i.e. by truncating the search tree at state s and replacing the subtree below s by an approximate value function. This approach has led to good performance in our game Connect4 and in other games by implementing Alpha-beta algorithm, but this was believed to be inconvenient in AlphaGo.

As mentioned in [2], because the board is 19×19 and moves are allowed into (almost) every empty square, the branching factor starts at 361, which is too daunting for regular alpha–beta search methods. In addition, it is difficult to write an evaluation function because control of territory is often very unpredictable until the endgame. Therefore the top programs avoid alpha–beta search and instead use **Monte Carlo rollouts**.

Comparing to the solution implemented in our program for Connect4 game, we are using heuristic evaluation function for guiding the search in a simple and primitive techniques, whereas in the AlphaGo approach developed in this paper they use an effective move selection and position evaluation functions based on trained deep neural networks, which requires more computation and uses an asynchronous multi-threaded search consisting of several CPUs and GPUs.

Using this new developed techniques, Connect4 would play better and get better performance because both AlphaGo and Connect4 are games of perfect information but with different order of search space, where Go is definitely larger.

5 Appendix

References

[1] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach, 3/e, Prentice Hall Series in Artificial Intelligence, 2010. ISBN-10: 0132071487 or 1292153962. Available: `https:\aima.cs.berkeley.edu`.

[2] Connect Four, Wikipedia: `https://en.wikipedia.org/wiki/Connect_Four`.

[3] Java implementation of algorithms from Russell and Norvig's Artificial Intelligence - A Modern Approach 3rd Edition. Alpha-Beta-Search: `https://github.com/aimacode/aima-java/blob/AIMA3e/aima-core/src/main/java/aima/core/search/adversarial/AlphaBetaSearch.java`

[4] James Dow Allen, *The Complete Book of Connect 4: History, Strategy, Puzzles*, Puzzle Wright Press, 2010. Expert Play in Connect-Four: `https://tromp.github.io/c4.html`

[5] Victor Allis, *A Knowledge-based Approach of Connect-Four*, The Game is Solved: White Wins. `http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf`

[6] D. Silver, A. Huang, et al. *Mastering the game of Go with Deep Neural Networks and Tree Search*, Nature 2016. `https://deepmind.com/research/publications/mastering-game-go-deep-neural-networks-tree-search`