Chapter 15
AlphaZero

Hongming Zhang and Tianyang Yu

Abstract In this chapter, we introduce combinatorial games such as chess and Go and take Gomoku as an example to introduce the AlphaZero algorithm, a general algorithm that has achieved superhuman performance in many challenging games. This chapter is divided into three parts: the first part introduces the concept of combinatorial games, the second part introduces the family of algorithms known as Monte Carlo Tree Search, and the third part takes Gomoku as the game environment to demonstrate the details of the AlphaZero algorithm, which combines Monte Carlo Tree Search and deep reinforcement learning from self-play.

Keywords AlphaZero · Monte Carlo Tree Search · Upper confidence bounds for trees · Self-play · Deep reinforcement learning · Deep neural network

15.1 Introduction

The AlphaZero (Silver et al. 2018, 2017a) algorithm is a generalized version of the AlphaGo Zero (Silver et al. 2017b) algorithm, which has achieved superhuman performance in the game of Go. Different from the initial AlphaGo (Silver et al. 2016) series such as AlphaGo Fan (defeated Fan Hui), AlphaGo Lee (defeated Lee Sedol), and AlphaGo Master (defeated Jie Ke), the AlphaZero algorithm is based solely on reinforcement learning from games via self-play (play against itself). It starts with random plays and without supervised learning from human expert games. There are two key parts in AlphaZero: (1) Monte Carlo Tree Search is used in self-play to collect data; (2) a deep neural network is used to learn from the data and predict the move selections and state values in Monte Carlo

H. Zhang (✉)
Peking University, Beijing, China
e-mail: zhanghongming@pku.edu.cn

T. Yu
Nanchang University, Nanchang, China

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_15

Tree Search. This general algorithm is suitable not only for the game of Go, but it also defeated world champion programs in the games of chess and shogi, which indicates its generality. In this chapter, we first introduce the concept of combinatorial games (including Go, chess, Gomoku, etc.) and implement the free-style Gomoku as an example, then explain the concrete steps in Monte Carlo Tree Search, and finally take Gomoku as the game environment to demonstrate the details of the AlphaZero algorithm. To help the readers' understanding, we provide the implementation of Gomoku game and the AlphaZero algorithm in the link: https://github.com/deep-reinforcement-learning-book/Chapter15-AlphaZero.

15.2 Combinatorial Games

Combinatorial game theory (CGT) (Albert et al. 2007) is a branch of research in mathematics and theoretical computer science that typically studies sequential games with perfect information. These types of games usually have the following characteristics:

- The game contains two players (Go, chess). One player's games (Sudoku, Solitaire) can also be regarded as combinatorial games that played between the game designer and the player. Games with more players are not considered combinatorial due to the social aspect of coalitions that may arise during play (Browne et al. 2012).
- The game does not contain any chance factors that will influence the outcome of the game such as a dice, etc.
- The game offers perfect information (Muthoo 1996), which means each player is perfectly informed of all the events that have previously occurred.
- The players perform actions in a turn-based manner, and the action space and state space are both finite.
- The game ends with finite time steps, results are divided into win and loss, and some games have a draw.

Many of the combinatorial games (Albert et al. 2007) including single-player games like Sudoku and Solitaire, two-player games like Hex, Go, and chess are classical problems for computer scientists to solve. Since IBM's DEEP BLUE (Campbell et al. 2002; Hsu 1999) beat grandmaster Gary Kasparov on chess, Go has become the standard yardstick for AI. Besides, there are also many other games like Othello, Amazons, Khet, Shogi, Chinese Checkers, Connect Four, Gomoku, etc., attracting people to find solutions with computing machines.

We take Gomoku as an example of combinatorial games and show some details about the code of Gomoku. Gomoku is also called "Gobang" or "five-in-a-row." It begins with an empty board and ends with a row of five stones (in a horizontal, vertical, or diagonal line) that indicates one player wins, or otherwise a draw. There

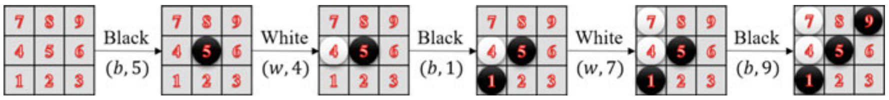


Fig. 15.1 A sequence of Gomoku gaming process on a 3 × 3 board. The "b" is shorten for "the black player" and the "w" is shorten for "the white player." (b, 5) means the black player puts the stone at the position 5. The black player wins the game in the end

are various sets of rules and most variations are based on either free-style Gomoku or standard Gomoku. Free-style Gomoku simply requires a row of five or more stones for a win. Standard Gomoku requires a row of exactly five stones for a win: rows of six or more, called overlines, do not count. In this game, we choose the free-style Gomoku as a demonstration.

We further simplify the board to have a size of 3 × 3 as an example for demonstration here. A row of three stones indicates winning (we may call it "three-in-a-row"). A sequence of game steps on this board are shown in Fig. 15.1.

The red numbers on the board are used for indexing different possible positions on the board, which can be used to represent the choice for each move. White and black circles are the game stones of two players. The gaming process can be represented as a sequence: ((b, 5), (w, 4), (b, 1), (w, 7), (b, 9)), where the "b" stands for "the black player" and the "w" stands for "the white player". The black player has a row of three stones in the end which means he wins the game, as shown in the last board state in Fig. 15.1. Recall the definition we mentioned before, this simplified Gomoku (or "three-in-a-row") satisfies all characteristics of combinatorial games: the game contains two players; the game does not contain any chance factors; the game offers perfect information; the players perform actions in a turn-based manner; the game ends with finite time steps.

Here, we implement the free-style Gomoku as an example:

- Define the game as `Board` class with the rule of the game implemented as some functions. Though we illustrated the rule of Gomoku with a simplified version before, we can define a standard five-in-a-row by passing the `n_in_row` variable with value 5.

```
class Board(object):
    '''
    board for the game
    '''
    def __init__(self, width, height, n_in_row): ... #
        Initialization function
    def move_to_location(self, move): ... # Transfer move
        denotation
    def location_to_move(self, location): ... # Transfer move
        denotation
    def do_move(self, move): ... # Update each move and exchange
        the current player
```

```
def has_a_winner(self): ... # The rule of Gomoku, to judge if
    someone wins
def current_state(self): ... # Generate board states as
    inputs of the network
...
```

- Each action on the board is denoted by a number like pictured in Fig. 15.1. This way makes it more convenient to build tree nodes in MCTS. But it is inconvenient to identify whether there is a row of five stones. So, we define the transition functions between coordinates and numbers. Coordinates are used to judge whether a player has a row of five stones, and numbers are used to build tree nodes in MCTS.

```
def move_to_location(self, move):
   '''
   Transfer number to coordinate
   3*3 board's moves like:
   6 7 8
   3 4 5
   0 1 2
   and move 5's location is (1,2)
   '''
   h = move // self.width
   w = move % self.width
   return [h, w]

def location_to_move(self, location):
   '''
   Transfer coordinate to number
   '''
   if len(location) != 2:
      return -1
   h = location[0]
   w = location[1]
   move = h * self.width + w
   if move not in range(self.width * self.height):
      return -1
   return move
```

- To find out if a player wins, a function is needed to judge if there is a line of five stones in a row or in a column or in a diagonal. The function `has_a_winner()` is shown:

```
def has_a_winner(self):
   '''
   Judge if there's a 5-in-a-row, and which player if so
   '''
   width = self.width
   height = self.height
```

```
states = self.states
n = self.n_in_row

moved = list(set(range(width * height)) -
    set(self.availables))
# Moves have been played
if len(moved) < self.n_in_row + 2:
   # Too few moves to get 5-in-a-row
   return False, -1

for m in moved:
   h, w = self.move_to_location(m)
   player = states[m]

   if (w in range(width - n + 1) and
         len(set(states.get(i, -1) for i in range(m, m +
             n))) == 1):
      # Judge if there's a 5-in-a-row in a line
      return True, player

   if (h in range(height - n + 1) and
         len(set(states.get(i, -1) for i in range(m, m + n
             * width, width))) == 1):
      # Judge if there's a 5-in-a-row in a column
      return True, player

   if (w in range(width - n + 1) and h in range(height - n
       + 1) and
         len(set(states.get(i, -1) for i in range(m, m + n
             * (width + 1), width + 1))) == 1):
      # Judge if there's a 5-in-a-row in a top right
          diagonal
      return True, player

   if (w in range(n - 1, width) and h in range(height - n
       + 1) and
         len(set(states.get(i, -1) for i in range(m, m + n
             * (width - 1), width - 1))) == 1):
      # Judge if there's a 5-in-a-row in a top left
          diagonal
      return True, player

return False, -1
```

15.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) (Browne et al. 2012) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. This method started a revolution

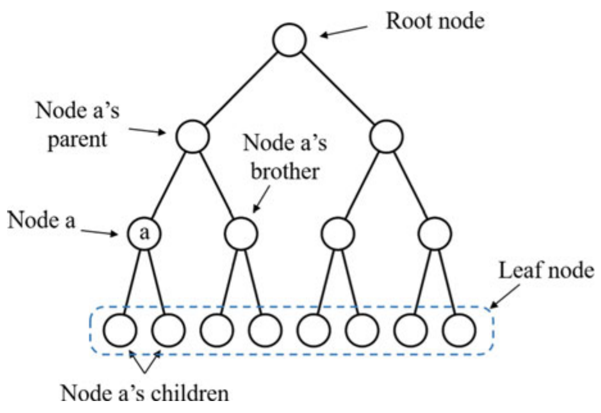


Fig. 15.2 A tree structure

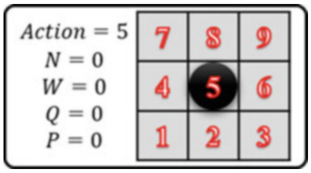


Fig. 15.3 A node with the state that a black stone on the board at the position 5. The action can be represented as $(b, 5)$ with $Action = 5$ by the black player. $N = 0$ means the visit count of this node is 0. W means the total rewards, Q means the average reward, P means the probability of taking the $Action = 5$. As the node has not been visited, these values are initialized with 0

in the fields of games and planning problems and pushed the performance such as computer Go to ever high levels.

Monte Carlo Tree Search includes two parts, the tree and the search methods. The tree is a kind of data structure (Fig. 15.2), it contains nodes connected by edges. Some important concepts include the root node, the parent and the child, the leaf node, etc. The node at the top of the tree is called the root node; the node above the current node is its parent, and the node below the current node is its child; a node that does not have child node is called a leaf node. Usually, besides the state and actions, the node in MCTS contains information about the visit count and average reward. In the AlphaZero algorithm, the tree also contains the probability distribution of the action for each state.

In the AlphaZero algorithm, each node contains the following information that is used to design search methods. (The node is illustrated in Fig. 15.3.)

- $Action$: action that is taken and reaches this node.
- N: the node's visit count. Initial value is zero, means the node has not been visited.
- W: the node's total reward which is used to calculate the average reward. Its initial value is zero.

- Q: the node's average reward which indicates the state value of the node. Its initial value is zero.
- P: the probability of taking the $Action$. It is the output by the policy network with the input of its parent node's state.

Before we go ahead, we have to mention an important aspect. Because the game is played by two players, there are two perspectives in a tree and the information at a node is either from the perspective of the black player or the perspective of the white player. For example, in Fig. 15.3, this node has a black stone on the board, so it should be the white player's turn to play the next move at this node. However, the information at this node is from the perspective of its parent node, which is the black player. It is the parent node that expands its child nodes and chooses an action to reach one of these child nodes, so the $Action$, N, W, Q, P at this node are all used by the black player. The black player takes $Action = 5$ to reach this node, and the information at the present time is $N = 0$, $W = 0$, $Q = 0$, $P = 0$. It is very important to have a clear understanding of the perspective of each node; otherwise, it will be confusing when we do the *backup* step in MCTS.

With the tree, the search methods aim to explore the decision space and estimate the state-action value function $Q^{\pi}(s, a)$ of the root node by some heuristic methods. Exploring from the root node to leaf nodes, obtaining the average reward of each action over multiple explorations, and finding the optimal trajectory in the tree. The action-value function without a discount factor can be expressed as follows (Couetoux et al. 2011):

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[\sum_{h=0}^{T-1} P(S_{h+1}|S_h, A_h)R(S_{h+1}|S_h, A_h)|S_0 = s, A_0 = a, A_h = \pi(S_h)\right]. \tag{15.1}$$

$Q^{\pi}(s, a)$ denotes the value function, i.e., the expected reward gathered when executing action a in state s and following policy π for all subsequent actions until arriving at a terminal state.

Usually, there are four steps in each tree search iteration: *select, expand, simulate, backup* (all these steps are conducted in the tree search, which indicates there is not a real move on the board).

- *Select*: actions are selected from the root node by a particular policy until reaching a leaf node.
- *Expand*: child nodes are added to the current leaf node.
- *Simulate*: a policy such as a random policy is used to simulate the process of playing stones until the end of the game to get a result: win, loss, or draw. Reward is given from the game and usually $+1$ for a win, -1 for a loss, 0 for a draw.
- *Backup*: the simulating result is backpropagated to update the information at each node that is selected in this iteration.

The most commonly used tree search algorithm is the Upper Confidence Bound in Tree (UCT) (Kocsis and Szepesvári 2006), which handles the exploration versus exploitation trade-off well. The Upper Confidence Bound (UCB) (Auer et al. 2002) (discussed in Chap. 2 Sect. 2.2.2) algorithm is a classical strategy to solve the multi-armed bandit problem, in which the agent has to choose among various gambling machines at each time step to maximize its payoffs. UCB chooses the next action at time t by the following policy:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]. \tag{15.2}$$

$Q_t(a)$ is the estimated action value which determines the exploitation. The square-root term determines the exploration. $N_t(a)$ is the number of times action a has been selected till time t and c is a positive real number that determines how much exploration needs to be done. There are some variations in the family of UCB algorithms, like UCB1, UCB1-NORMAL, UCB1-TUNED, and UCB2 (Auer et al. 2002).

UCT is the version that implements UCB1 in the Tree. Each time, a child node with the largest UCT value is selected which is defined as follows:

$$\text{UCT} = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}}. \tag{15.3}$$

Here, n is the number of visits to the current node, n_j is the number of visits to its child node j, and $C_p > 0$ is a constant that controls the exploration based on specific problems. The average reward term $\overline{X}_j$ encourages the exploitation of higher-reward actions, while the square-root term $\sqrt{\frac{2\ln n}{n_j}}$ encourages the exploration of less-visited actions.

UCT algorithm addresses the exploration versus exploitation trade-off in each state of the tree search space and has revolutionized quite a few large-scale RL problems such as board game Hex, Go, and real-time games like Atari. Levente Kocsis and Csaba Szepesv́ari (Kocsis and Szepesvári 2006) proved that: Consider a finite-horizon MDP with rewards scaled to lie in the interval [0, 1]. Let the horizon of the MDP be D, and the number of actions per state be K. Consider algorithm UCT such that the bias terms of UCB1 are multiplied by D. Then the bias of the estimated expected payoff, $\overline{X}_n$, is $O(\frac{\log n}{n})$. Further, the failure probability at the root converges to zero at a polynomial rate as the number of episodes grows to infinity. It indicates that, as the number of explored samples increases, the UCT algorithm can guarantee the tree search converges to the optimal solution.

In the AlphaZero algorithm, the step of *simulate* is discarded and the deep neural network is used to output the result directly. So, there are three key steps as follows, and the process is pictured in Fig. 15.4.

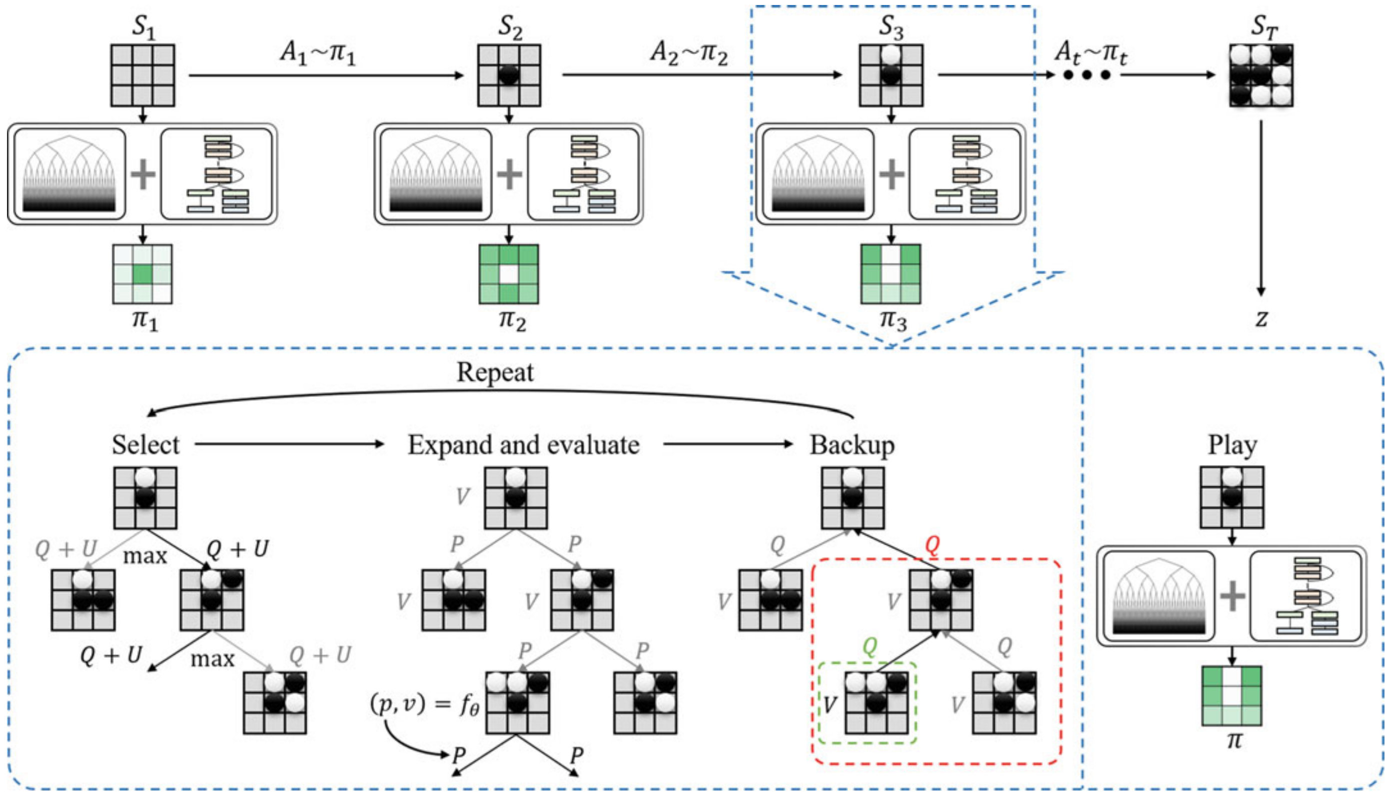


Fig. 15.4 MCTS in the AlphaZero algorithm. Each time before a stone is placed on the real board, MCTS will be repeated for many times. It first selects actions from the root node and reaches a leaf node, then the leaf node is expanded and evaluated, finally the *backup* step is taken to update nodes' information

- *Select*: actions are selected from the root node by a particular policy until reaching a leaf node.
- *Expand and evaluate*: child nodes are added to the current leaf node. The probability of each action and the value of the state are evaluated directly by the policy network and the value network. A threshold value is usually applied to judge if the node is supposed to be expanded, for saving the computing resources without losing the effectiveness of the algorithm. In our implementation, we ignore this threshold and expand the node as long as it reaches a leaf node.
- *Backup*: each time after expanding and evaluating, the result is backpropagated to update the information at the nodes that are selected in the current iteration. If the leaf node is not the terminal of the game, the result cannot be given by the game and it is output by the deep neural network; otherwise, it is given by the game directly.

In the step of *select*, the action is selected by the formula $a = \arg\max_a(Q(s, a) + U(s, a))$. $Q(s, a) = \frac{W}{N}$ encourages the exploitation of higher-reward actions. $U(s, a) = c_{puct} P(s, a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ encourages the exploration of less-visited actions, c_{puct} is a parameter determining the exploration scale, which is 5 in the AlphaZero algorithm.

In the step of *expand and evaluate*, the policy network outputs a probability $p(s, a)$ for each action, the value network outputs a value v indicating the state value for the current state s. The $p(s, a)$ is used to calculate $U(s, a)$ in the step of *select*, $U(s, a) = c_{puct} P(s, a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$. The v is used to calculate W in the step of *backup*, $W(s, a) = W(s, a) + v$. The probabilities and values output by the neural networks may not be accurate at the beginning, but they will be more accurate gradually during the training process.

In the step of *backup*, information at each selected node is updated, $N(s, a) = N(s, a) + 1$, $W(s, a) = W(s, a) + v$, $Q(s, a) = \frac{W(s,a)}{N(s,a)}$.

Some core code is shown as follows:

- The process of Monte Carlo Tree Search is defined as a `MCTS` class, it contains the whole tree and the tree search function `_playout()`:

```
class MCTS(object):
    '''
    An implementation of Monte Carlo Tree Search.
    '''
    def __init__(self, policy_value_fn,action_fc,evaluation_fc,
        is_selfplay,c_puct, n_playout): ... # Init function
    def _playout(self, state): ... # The process of tree search
```

- The node in the tree is defined as a `TreeNode` class with above steps: *select, expand and evaluate, backup*.

```
class TreeNode(object):
    '''
    A node in the tree.
    Each node keeps track of its own value Q, prior
        probability P, and
    its visit-count-adjusted prior score u.
    '''
    def __init__(self, parent, prior_p): ... # Init function
    def select(self, c_puct): ... # Choose action
    def expand(self, action_priors, add_noise): ...# Expand
        the node, evaluate each action and state
    def update(self, move): ... # Update node after expanding
        and evaluating
    ...
```

- The function `select()` corresponds to the step of *select*:

```
def select(self, c_puct):
    '''
    Select an action among children that gives maximum action
        value Q plus bonus u(P).
    Return: A tuple of (action, next_node)
    '''
    return max(self._children.items(),
               key=lambda act_node: act_node[1].get_value(c_puct))
```

- The function `expand()` here corresponds to the step of *expand and evaluate*. Besides, we add some Dirichlet noises at the nodes for random exploration:

```
def expand(self, action_priors, add_noise):
    '''
    Expand tree by creating new children.
    action_priors: a list of tuples of actions and their prior
        probability
       according to the policy function.
    '''
    if add_noise:
       action_priors = list(action_priors)
       length = len(action_priors)
       dirichlet_noise = np.random.dirichlet(0.3 *
           np.ones(length))
       for i in range(length):
          if action_priors[i][0] not in self._children:
             self._children[action_priors[i][0]] =
                 TreeNode(self,
                0.75 * action_priors[i][1] + 0.25 *
                    dirichlet_noise[i])
    else:
        for action, prob in action_priors:
            if action not in self._children:
                self._children[action] = TreeNode(self, prob)
```

- The function `update_recursive()` corresponds to the step of *backup*:

```
def update_recursive(self, leaf_value):
    '''
    Like a call to update(), but applied recursively for all
        ancestors.
    '''
    # If it is not root, this node's parent should be updated
        first.
    if self._parent:
        self._parent.update_recursive(-leaf_value)
        # Every step for recursive update.
        # We should change the perspective by the way of taking
            the opposite value
    self.update(leaf_value)

def update(self, leaf_value):
    '''
    Update node values from leaf evaluation.
    leaf_value: the value of subtree evaluation from the
        current player's
      perspective.
    '''
    self._n_visits += 1
    # update visit count
    self._Q += 1.0 * (leaf_value - self._Q) / self._n_visits
    # Update Q, a running average of values for all visits.
    # there is just: (v-Q)/(n+1)+Q =
        (v-Q+(n+1)*Q)/(n+1)=(v+n*Q)/(n+1)
```

- The tree search function `_playout()` in the class `MCTS` executes the three steps iteratively: *select, expand and evaluate, backup*.

```
def _playout(self, state):
    '''
    Run a single MCTS from the root to the leaf, get a value at
    the leaf and propagate it back through its parents.
    '''
    node = self._root
    # Select
    while(1):
            if node.is_leaf():
                break
            action, node = node.select(self._c_puct)
            state.do_move(action)
        # Evaluate and expand
        action_probs, leaf_value =
             self._policy_value_fn(state,self._action_fc,self._evaluation_fc)
        end, winner = state.game_end()
        if not end:
            node.expand(action_probs,add_noise=self._is_selfplay)
        else:
            if winner == -1: # draw
                leaf_value = 0.0
            else:
                leaf_value = (
                    1.0 if winner == state.get_current_player() else -1.0
                )
        # Backup
        node.update_recursive(-leaf_value)
```

15.4 AlphaZero: A General Algorithm for Board Games

Generally, the AlphaZero algorithm is suitable for all kinds of combinatorial games, such as Go, chess, Shogi, and so on. Here, we take Gomoku with the free-style rule described in Sect. 15.2 as an example, to introduce the details of the AlphaZero algorithm. Gomoku is a turn-based game with simple rules, which is suitable for demonstration as the game itself is not the focus. We further simplify the game board to have size 3×3 as an example, and a row of three stones indicates winning as we mentioned before. In addition, it is necessary to mention that the AlphaZero algorithm generalizes the AlphaGo Zero algorithm, so the two algorithms are very similar. In our implementation, we compare both methods.

We will demonstrate the details of the AlphaZero algorithm in this section for better understanding. The whole algorithm can be split into two parts: (1) self-play process using Monte Carlo Tree Search for data collection; (2) the deep neural network for training. The schematic is shown in Fig. 15.5.

First, we execute the self-play process with MCTS for collecting data. In order to demonstrate the tree search to the end of the game with relatively short paragraphs, we assume the game begins from the state as shown in Fig. 15.6 (normally, the game is started from an empty board). It is the white player's turn to play a stone now.

We build a tree from this node and start the MCTS process following the three steps: *select, expand and evaluate, backup*. Now, there is only one node in the tree. This node is a root node since it is at the top of the tree, and it is also a leaf node as it does not have child node. It means we have reached a leaf node and the step of *select* is done. So, the node needs to be expanded, it turns to the second step: *expand and evaluate*. In Fig. 15.7, the node is expanded, and the probability of each action is given by the policy network, with the input of the current board state.

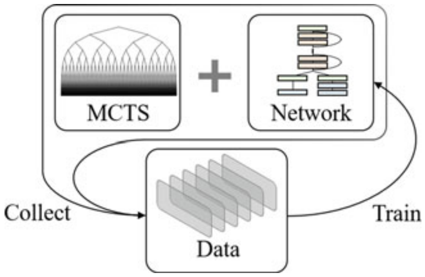


Fig. 15.5 Algorithm schematic. In the AlphaZero algorithm, a cycle is formed among MCTS, the data, and the network. MCTS is used to generate data, the data is used to improve the precision of the network, the more accurate network is used in MCTS to generate higher quality data

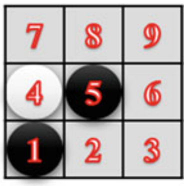


Fig. 15.6 Board state. The size of the board is 3 × 3. In this state, it is the white player's turn to play

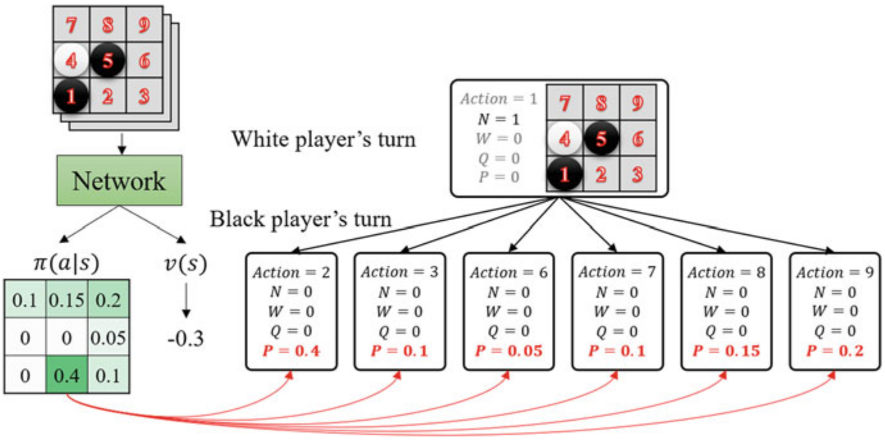

Network
$\pi(a|s)$
$v(s)$
0.1 0.15 0.2
0 0 0.05
0 0.4 0.1
-0.3
White player's turn
Black player's turn
Action = 1 N = 1 W = 0 Q = 0 P = 0
Action = 2 N = 0 W = 0 Q = 0 P = 0.4
Action = 3 N = 0 W = 0 Q = 0 P = 0.1
Action = 6 N = 0 W = 0 Q = 0 P = 0.05
Action = 7 N = 0 W = 0 Q = 0 P = 0.1
Action = 8 N = 0 W = 0 Q = 0 P = 0.15
Action = 9 N = 0 W = 0 Q = 0 P = 0.2

Fig. 15.7 *Expand and evaluate* at the root node. All the available actions are expanded, and the probabilities $\pi(a|s)$ are given by the network

The last step is *backup*. As it is a root node now, we do not need to backup W and Q (used to judge if we should go to this node) and only need to update the visit count N. $N = 0$ is changed to $N = 1$, and the tree search iteration has been finished once.

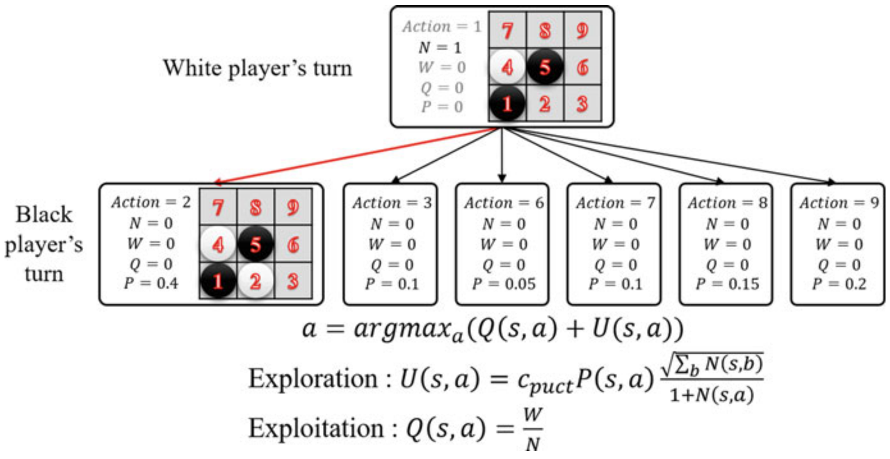


Fig. 15.8 *Select* at the root node. The white player selects $Action = 2$ $(w, 2)$ and reaches a leaf node. In this node, it is the black player's turn to play

At each time we execute a tree search iteration, we will begin from the root node. So, the second tree search process also starts from the root node. This time, the root node has child nodes, which means it is not a leaf node. The action is selected by the formula $a = \arg\max_a(Q(s,a)+U(s,a))$, $Q(s,a) = \frac{W}{N}$, $U(s,a) = c_{puct}P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$. Here, the action chosen by the white player is 2 $(w, 2)$, and it reaches a new node. This new node is a leaf node and it is the black player's turn at this node (Fig. 15.8).

We expand and evaluate this leaf node. It is the same as the first time: the node is expanded, and the probability of each action is given by the policy network (Fig. 15.9).

Then, it is time to backup. Now there are two nodes; we first update the current node and then the preceding node. Both of them follow the same rules for updating in the *backup* step: $N(s,a) = N(s,a) + 1$, $W(s,a) = W(s,a) + v(s)$, $Q(s,a) = \frac{W(s,a)}{N(s,a)}$. It should be noted that there are two perspectives in the tree: black and white. We should be careful about the perspectives and always update the value in the current player's perspective. For example, in Fig. 15.10, the value from the value network is $v(s) = -0.1$, it is from the perspective of the black player. When updating the information that belongs to the white player, it needs to be reversed, i.e., $v(s) = 0.1$. So, we get $N = 1$, $W = 0.1$, $Q = 0.1$.

Then we return to its parent node. Like in the first tree search process, as the current state is a root node, we do not need to backup W and Q here and only need to update the visit time N. So, let $N = 2$ and we have done another tree search as shown in Fig. 15.11.

The third tree search process also starts from the root node. With the formula $a = \arg\max_a(Q(s,a) + U(s,a))$ and the current information in the tree, the action chosen by the white player is 2 $(w, 2)$, and the black player chooses the action

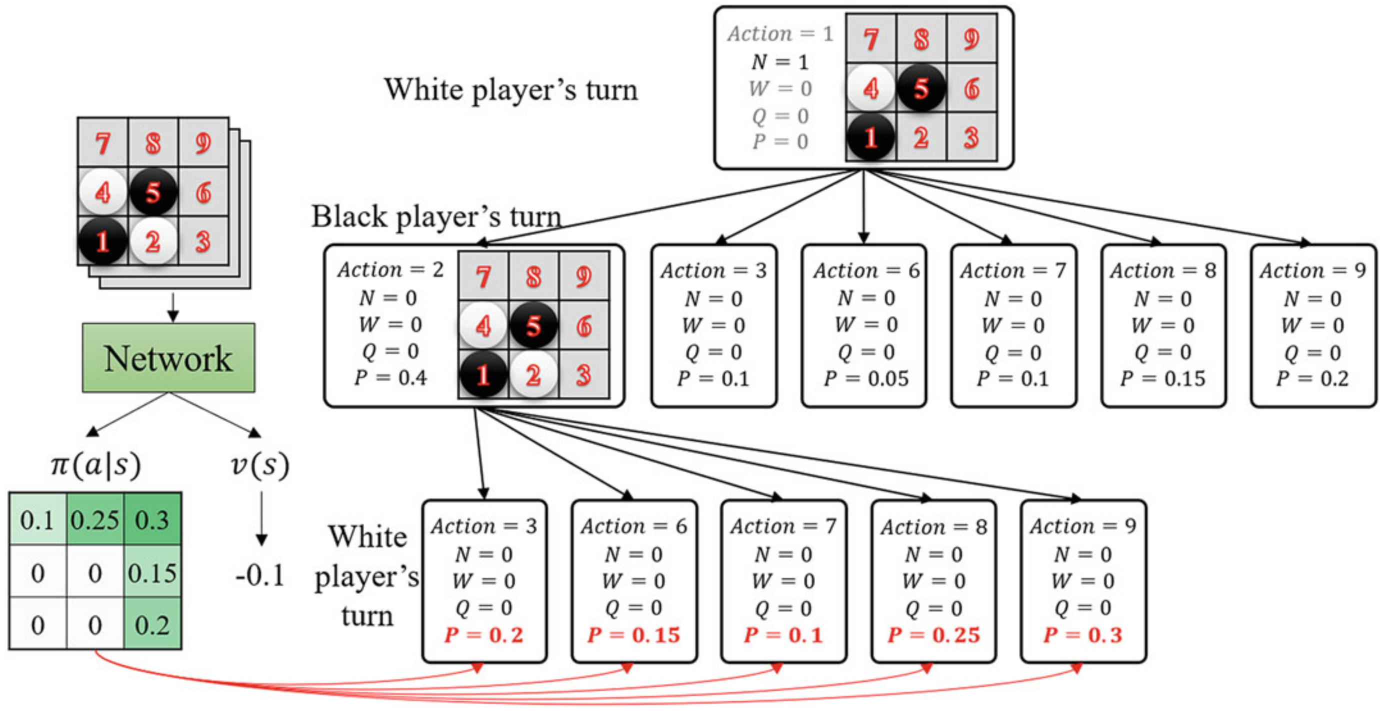

White player's turn
Action = 1
N = 1
W = 0
Q = 0
P = 0
7 8 9
4 5 6
1 2 3
Black player's turn
Action = 2
N = 0
W = 0
Q = 0
P = 0.4
Action = 3, N = 0, W = 0, Q = 0, P = 0.1
Action = 6, N = 0, W = 0, Q = 0, P = 0.05
Action = 7, N = 0, W = 0, Q = 0, P = 0.1
Action = 8, N = 0, W = 0, Q = 0, P = 0.15
Action = 9, N = 0, W = 0, Q = 0, P = 0.2
Network
π(a|s)
v(s)
-0.1
0.1 0.25 0.3
0 0 0.15
0 0 0.2
White player's turn
Action = 3, N = 0, W = 0, Q = 0, P = 0.2
Action = 6, N = 0, W = 0, Q = 0, P = 0.15
Action = 7, N = 0, W = 0, Q = 0, P = 0.1
Action = 8, N = 0, W = 0, Q = 0, P = 0.25
Action = 9, N = 0, W = 0, Q = 0, P = 0.3

Fig. 15.9 *Expand and evaluate* at the new node. All the available actions are expanded, and the probabilities $\pi(a|s)$ are given by the network

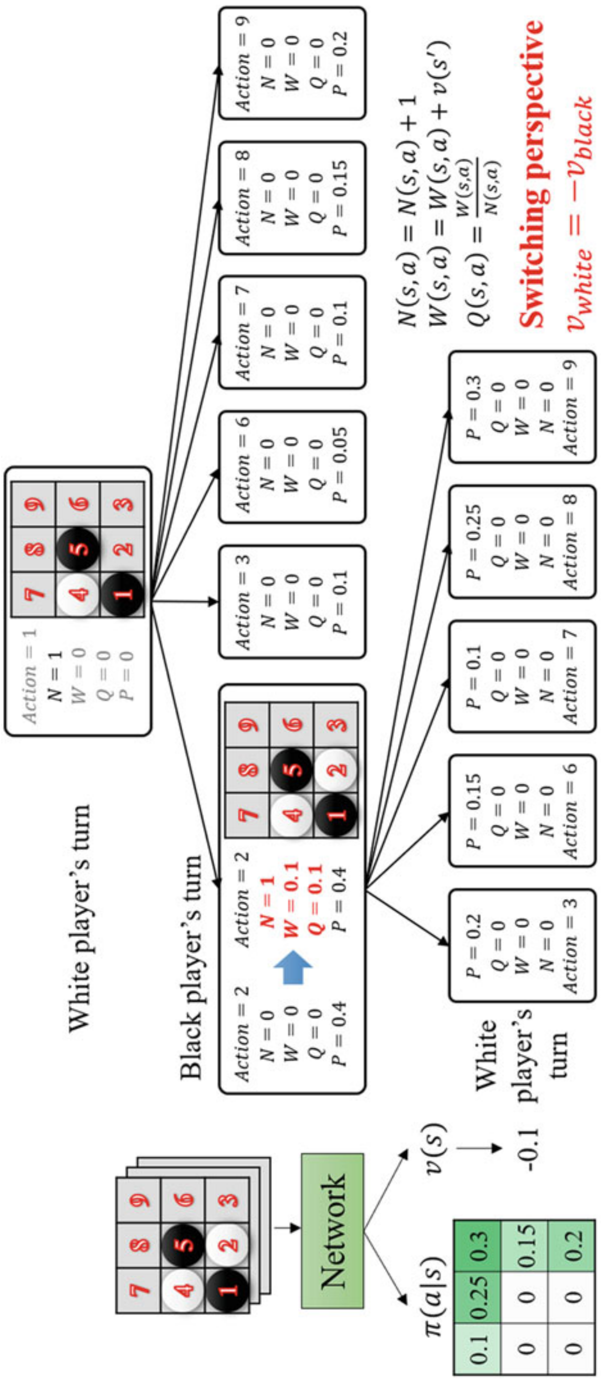

White player's turn
Action = 1
N = 1
W = 0
Q = 0
P = 0
Black player's turn
Action = 2
N = 0
W = 0
Q = 0
P = 0.4
Action = 2
N = 1
W = 0.1
Q = 0.1
P = 0.4
Action = 3, N = 0, W = 0, Q = 0, P = 0.1
Action = 6, N = 0, W = 0, Q = 0, P = 0.05
Action = 7, N = 0, W = 0, Q = 0, P = 0.1
Action = 8, N = 0, W = 0, Q = 0, P = 0.15
Action = 9, N = 0, W = 0, Q = 0, P = 0.2
White player's turn
Action = 3, N = 0, W = 0, Q = 0, P = 0.2
Action = 6, N = 0, W = 0, Q = 0, P = 0.15
Action = 7, N = 0, W = 0, Q = 0, P = 0.1
Action = 8, N = 0, W = 0, Q = 0, P = 0.25
Action = 9, N = 0, W = 0, Q = 0, P = 0.3
Network
π(a|s)
v(s)
-0.1
0.1 0.25 0.3
0 0 0.15
0 0 0.2
N(s,a) = N(s,a) + 1
W(s,a) = W(s,a) + v(s')
Q(s,a) = W(s,a)/N(s,a)
Switching perspective
v_white = −v_black

Fig. 15.10 *Backup* at the new node. The information at the current node is updated and Q should be updated in the white player's perspective: $N = 1$, $W = 0.1$, $Q = 0.1$

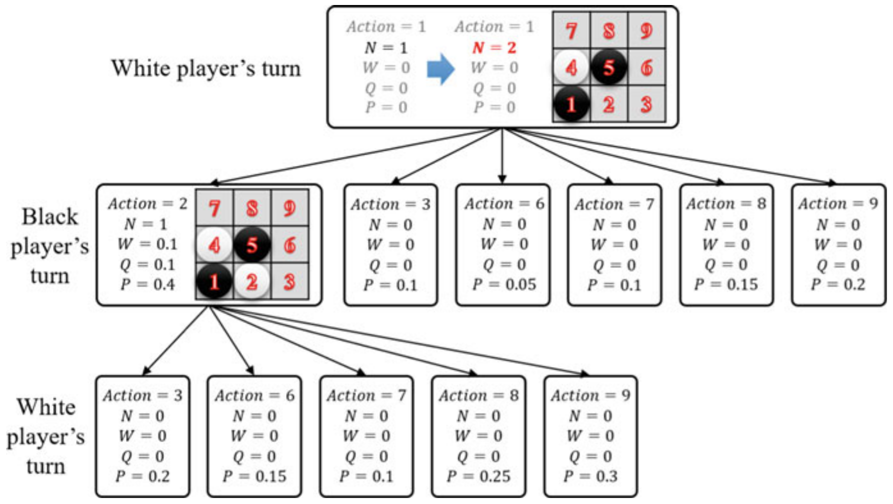


Fig. 15.11 *Backup* at the root node. N is updated to 2, W and Q are not needed to update

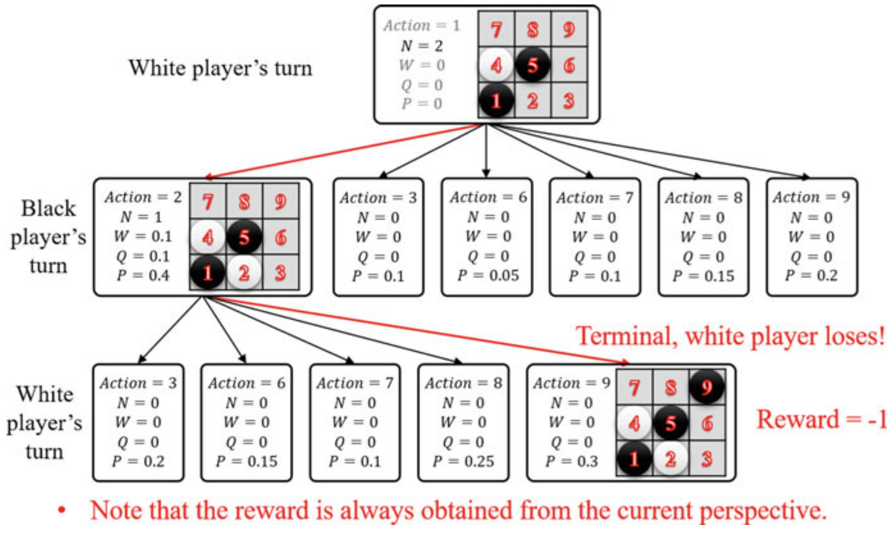


Fig. 15.12 *Expand and evaluate* at the terminal node. Since the game is end at this node, no node will be expanded and the reward can be obtained from the game directly. So, the policy network and the value network are not used here

9 $(b, 9)$. As shown in Fig. 15.12, the *select* step leads to a new node that is the end of the game. This time, for the step of *expand and evaluate*, the node will not be expanded and the value v can be obtained from the game directly. So, the value network is not used to evaluate the state and the policy network is not used to output the probability of each action.

Then, it is time to backup and there are three nodes in the trajectory. As we mentioned before, the nodes are updated recursively from the leaf node to the root node, $N(s,a) = N(s,a) + 1$, $W(s,a) = W(s,a) + v(s)$, $Q(s,a) = \frac{W(s,a)}{N(s,a)}$. Moreover, the perspective of each node should also be switched, which means $v_{white} = -v_{black}$. In this game, the black player chooses 9 $(b, 9)$ and reaches a new node. Now it is the white player's turn to choose an action, but unluckily the game is over and the white player loses the game. So the $reward = -1$ is from the perspective of the white player, that is to say $v_{white} = -1$. When we update the information at this node, as we mentioned before, these information is used by the black player to choose $Action = 9$ to reach this node, so the value for this node should be $v_{black} = -v_{white} = 1$, and other information is based on it, $N = 1$, $W = 1$, $Q = 1$. The information at other nodes are updated in the same way.

After the *backup* step, the renewed tree is shown in Fig. 15.13. The root node has been visited three times and the information at each visited node is updated.

We have demonstrated three times MCTS above. After the tree search is conducted 400 times as shown in Fig. 15.14 (in the AlphaGo Zero algorithm, the number is 1600; in the AlphaZero algorithm, the number is 800), the tree has grown much larger and the estimated values are more accurate.

After the MCTS, a stone can be placed on the real board now. The action is chosen by calculating the probability related to the visit count rather than the outputs of the policy network: $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{N(s)^{1/\tau}-1} = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$, where $\tau \to 0$ is a

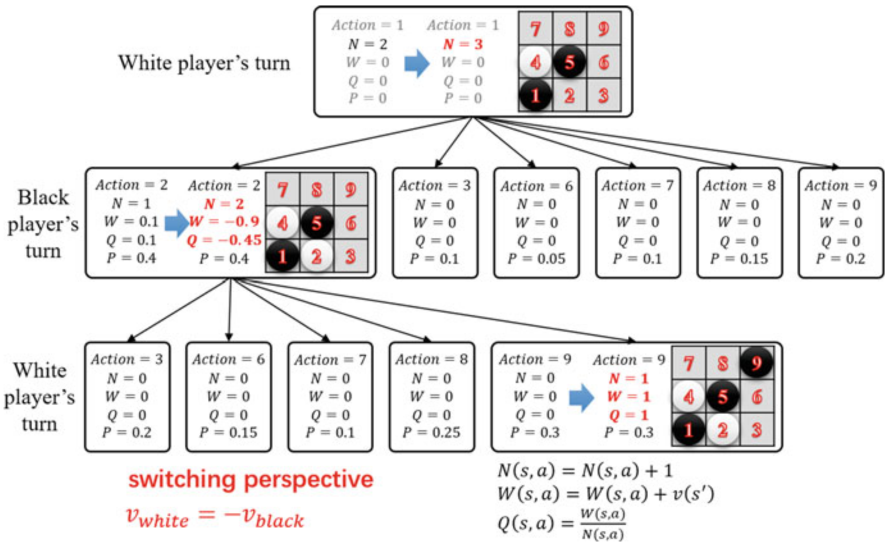

White player's turn
Action = 1
N = 2
W = 0
Q = 0
P = 0
Action = 1
N = 3
W = 0
Q = 0
P = 0
Black player's turn
Action = 2
N = 1
W = 0.1
Q = 0.1
P = 0.4
Action = 2
N = 2
W = −0.9
Q = −0.45
P = 0.4
Action = 3 N = 0 W = 0 Q = 0 P = 0.1
Action = 6 N = 0 W = 0 Q = 0 P = 0.05
Action = 7 N = 0 W = 0 Q = 0 P = 0.1
Action = 8 N = 0 W = 0 Q = 0 P = 0.15
Action = 9 N = 0 W = 0 Q = 0 P = 0.2
White player's turn
Action = 3 N = 0 W = 0 Q = 0 P = 0.2
Action = 6 N = 0 W = 0 Q = 0 P = 0.15
Action = 7 N = 0 W = 0 Q = 0 P = 0.1
Action = 8 N = 0 W = 0 Q = 0 P = 0.25
Action = 9
N = 0
W = 0
Q = 0
P = 0.3
Action = 9
N = 1
W = 1
Q = 1
P = 0.3
switching perspective
$v_{white} = -v_{black}$
$N(s,a) = N(s,a) + 1$
$W(s,a) = W(s,a) + v(s')$
$Q(s,a) = \frac{W(s,a)}{N(s,a)}$

Fig. 15.13 The tree after the *backup* step. In the process of the third MCTS, the *backup* step updates the information of the three visited nodes recursively. Be careful, because two players are in a single tree and $v_{white} = -v_{black}$, the information should be updated from the right perspective

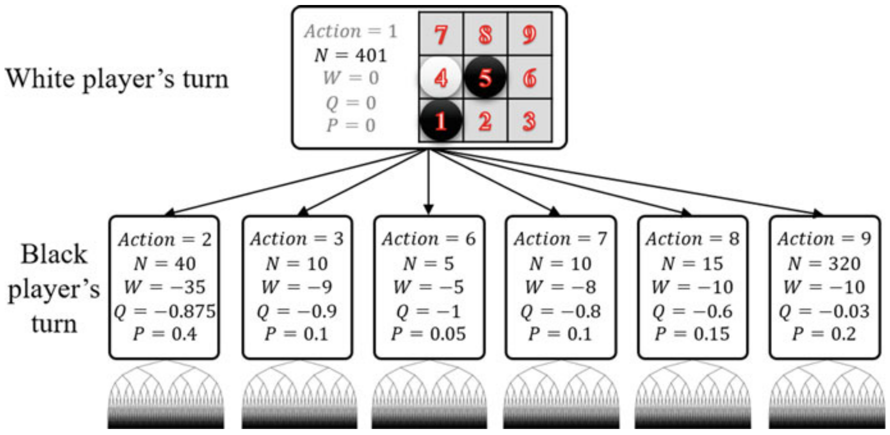
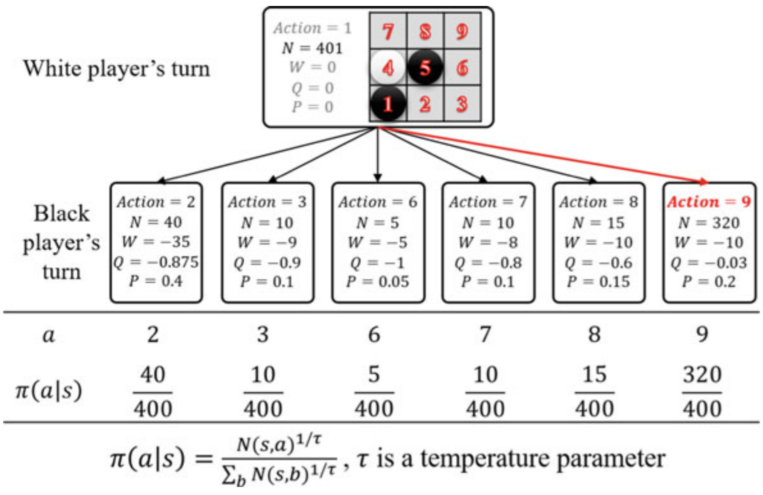

White player's turn
Action = 1
N = 401
W = 0
Q = 0
P = 0
7 8 9
4 5 6
1 2 3
Black player's turn
Action = 2, N = 40, W = −35, Q = −0.875, P = 0.4
Action = 3, N = 10, W = −9, Q = −0.9, P = 0.1
Action = 6, N = 5, W = −5, Q = −1, P = 0.05
Action = 7, N = 10, W = −8, Q = −0.8, P = 0.1
Action = 8, N = 15, W = −10, Q = −0.6, P = 0.15
Action = 9, N = 320, W = −10, Q = −0.03, P = 0.2

Fig. 15.14 The whole tree after searching 400 times. As the first MCTS process is started from the step of *expand and evaluate* and it does not select a child node, the sum of the visit counts of its child nodes is 400 and the root node's visit count is 401


White player's turn
Action = 1
N = 401
W = 0
Q = 0
P = 0
7 8 9
4 5 6
1 2 3
Black player's turn
Action = 2, N = 40, W = −35, Q = −0.875, P = 0.4
Action = 3, N = 10, W = −9, Q = −0.9, P = 0.1
Action = 6, N = 5, W = −5, Q = −1, P = 0.05
Action = 7, N = 10, W = −8, Q = −0.8, P = 0.1
Action = 8, N = 15, W = −10, Q = −0.6, P = 0.15
Action = 9, N = 320, W = −10, Q = −0.03, P = 0.2

a	2	3	6	7	8	9
$\pi(a\|s)$	$\frac{40}{400}$	$\frac{10}{400}$	$\frac{5}{400}$	$\frac{10}{400}$	$\frac{15}{400}$	$\frac{320}{400}$

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, \tau \text{ is a temperature parameter}$$

Fig. 15.15 Play a move on the board. After searching 400 times, an action is selected according to $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$. Here, the white player selects 9 $(w, 9)$

temperature parameter, $b \in A$ denotes the available action at state s. Here, the selected action is 9 $(w, 9)$ as shown in Fig. 15.15.

The temperature parameter is used to control the exploration. If $\tau = 1$, it selects moves proportionally to their visit counts in MCTS, this means a high-level exploration and ensures a diverse set of positions are encountered. If $\tau \to 0$, it means a low exploration and selects the move with maximum visit count. In the AlphaZero and AlphaGo Zero algorithm, when they perform the self-play process to collect data, the temperature is set to $\tau = 1$ for the first 30 moves (12 moves in our implementation), and $\tau \to 0$ for the remainder of the game. When playing a real game with an opponent, the temperature is set to $\tau \to 0$ all the time.

The white stone has been put at the position 9 $(w, 9)$ on the board, so the root node in the tree will be changed to the child node and MCTS will go on from this new root node. Other sibling nodes and its parent node will be discarded to prune the tree and save memory (Fig. 15.16).

When the game is over on the game board, we get the data and the result (Fig. 15.17).

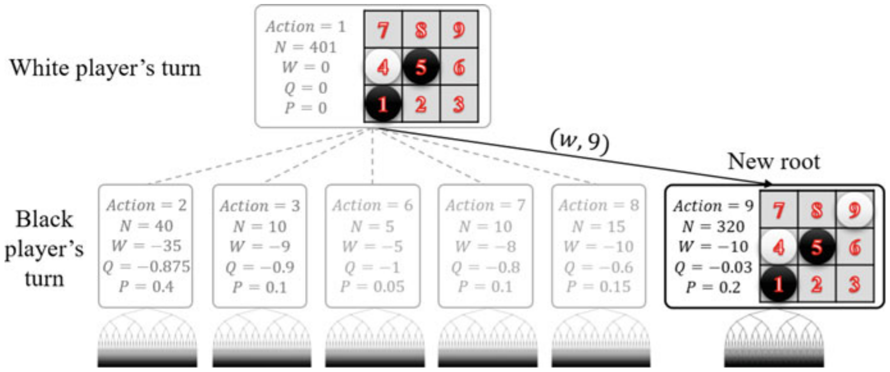


Fig. 15.16 The new root node. The nodes that under the new root node will be maintained, and other nodes will be discarded

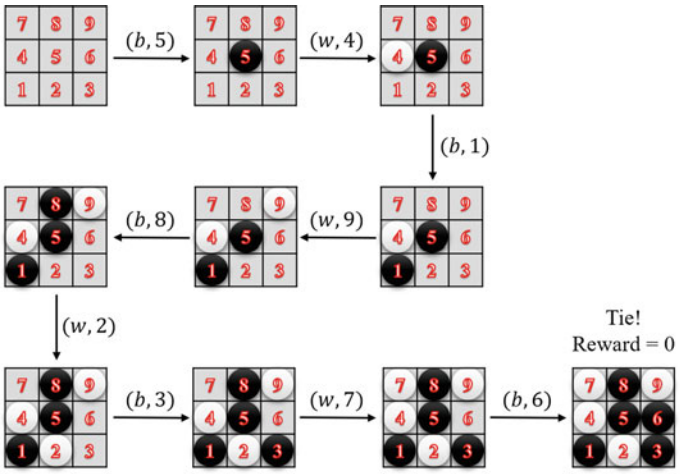
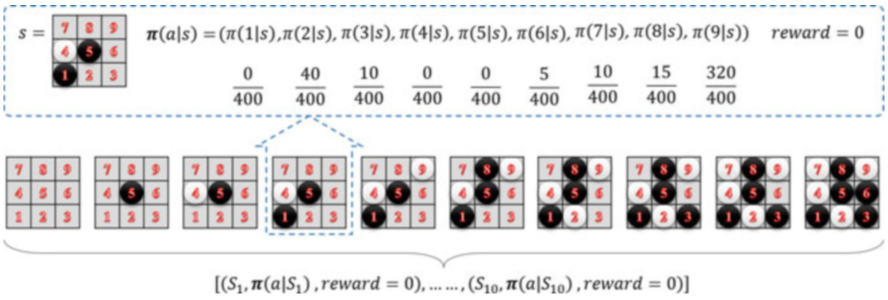


Fig. 15.17 The game data. All states in the game will be saved and labeled with probabilities $\pi(a|s)$ and value $v(s)$



s =
π(a|s) = (π(1|s), π(2|s), π(3|s), π(4|s), π(5|s), π(6|s), π(7|s), π(8|s), π(9|s))
reward = 0
0/400 40/400 10/400 0/400 0/400 5/400 10/400 15/400 320/400
[(S1, π(a|S1), reward = 0), ……, (S10, π(a|S10), reward = 0)]

Fig. 15.18 Data with labels. The probability of the action is according to $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$, and the value $v(s)$ is from the result of the game: $+1$ for a win, -1 for a loss, 0 for a draw

The probability for each action is: $\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, \tau = 1$. Be careful that the probabilities here are related to visit counts; this is the key point that combines the MCTS self-play and policy network training. As the result of this game is a draw, the labels for the value network here are 0 for all these data (Fig. 15.18).

Now, the data is available by the self-play process using Monte Carlo Tree Search, the next part is to apply the deep neural network for data training. In the training process, the data is first transformed into some feature planes. Each feature plane composes binary values indicating the presence of the player's stones, with one set of planes for the current player and another set of planes for the opponent's stones. These planes are concatenated together in order with history features included. Then, we augment the data in the same way as in the AlphaGo Zero algorithm: Since the rules of Go and Gomoku are both invariant to rotation and reflection, the data are augmented by rotation and reflection before training. And during the MCTS, board positions are randomly rotated or reflected before being evaluated by the neural network, which can average different biases. However, in the AlphaZero algorithm, the trick is not used since some games' rules are not invariant to rotation and reflection. With more and more sampled data collected, the network is trained to be more accurate for estimation.

We use `ResNet` (He et al. 2016) as the network structure (Fig. 15.19) which is the same as the AlphaGo Zero algorithm. The network's inputs are the game states, the outputs are the probabilities of actions and the values of the states. The network can be denoted as $(\boldsymbol{p}, v) = f_\theta(s)$ and the data is $(s, \boldsymbol{\pi}, r)$. The loss function l combines the cross-entropy losses over the actions' probability distribution, mean-squared error over the state value, and the L2 weight regularization over the parameters. The concrete formula is $l = (r - v)^2 - \boldsymbol{\pi}^T \log \boldsymbol{p} + c||\theta||^2$, where c is a parameter controlling the level of regularization.

There are some details about the timing of updating the model. In the AlphaGo Zero algorithm, the new model will play with the current best model for 400 games; if the new one wins by a margin of 55%, it becomes a better model. By contrast, in the version of the AlphaZero algorithm, it does not play against previous models and the parameters are updated continuously. These are alternative approaches with

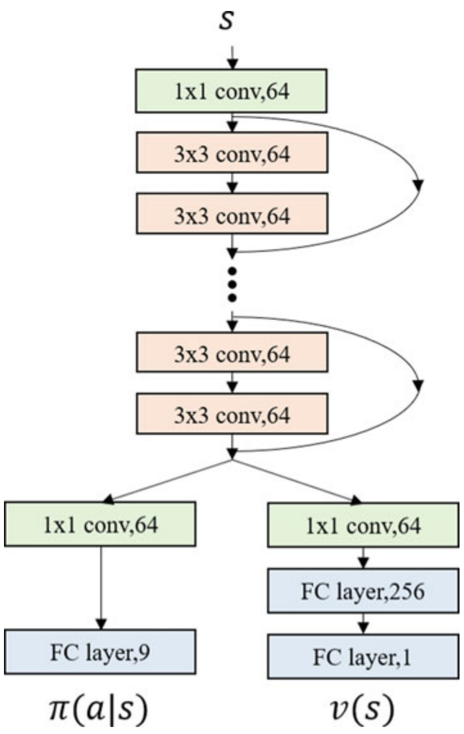


Fig. 15.19 Network structure. The structure is the same with the AlphaGo Zero algorithm. `ResNet` is used as the backbone and two heads output the probability distribution and state value separately

different performances, and we implement our code in the AlphaGo Zero's manner in this project to make the training process more stable. Besides, if you want to train it faster, you can collect data in a parallel way using multi-process and do the tree search process asynchronously as the original paper does. In the parallel way (Fig. 15.20), there are many processes executing at the same time: the parameters of the neural network are trained with the latest self-play data, the new self-play data are generated from the best model, the latest model is continually evaluated (in AlphaGo Zero), all these components are executed simultaneously.

Then, with a stream of new data saved in a buffer and with this neural network to learn a more accurate policy head and value head, we will get a powerful Gomoku AI after iterating the MCTS and network training for sufficient times.

We finally trained a Gomoku model in a parallel way with the free-style rule (a row of five or more stones for a win) on a 11×11 board. Some specific parameters are presented in Table 15.1. A model on a 15×15 board is also trained successfully with the same parameters, which indicates the generality and stability of the AlphaZero algorithm.

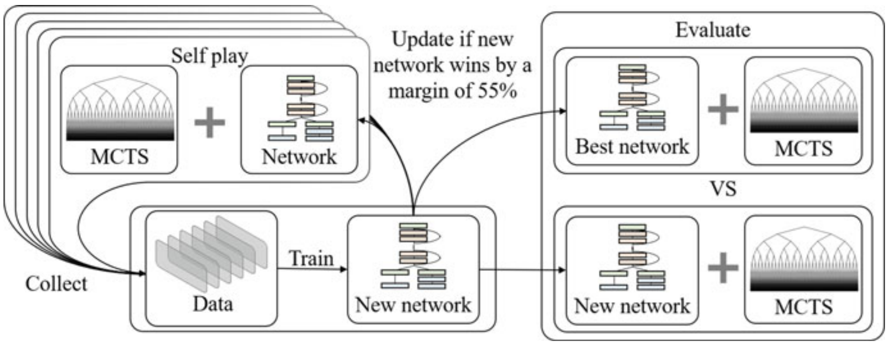


Fig. 15.20 Parallel training structure

Table 15.1 Comparison of parameters

Parameters setting	Gomoku	AlphaGo Zero	AlphaZero
c_{puct}	5	5	5
MCTS times	400	1600	800
Residual blocks	19	19/39	19/39
Batch size	512	2048	4096
Learning rate	0.001	Annealed	Annealed
Optimizer	Adam	SGD with momentum	SGD with momentum
Dirichlet noise	0.3	0.03	0.03
Weight of noise	0.25	0.25	0.25
$\tau = 1$ for the first n moves	12	30	30

References

Albert M, Nowakowski R, Wolfe D (2007) Lessons in play: an introduction to combinatorial game theory. CRC Press, Boca Raton

Auer P, Cesa-Bianchi N, Fischer P (2002) Finite-time analysis of the multiarmed bandit problem. Mach Learn 47(2–3):235–256

Browne CB, Powley E, Whitehouse D, Lucas SM, Colton S (2012) A survey of Monte Carlo tree search methods. IEEE Trans Comput Intell Ai Games 4(1):1–43

Campbell M, Hoane Jr AJ, Hsu FH (2002) Deep blue. Artif. Intell. 134(1–2):57–83

Couetoux A, Milone M, Brendel M, Doghmen H, Sebag M, Teytaud O (2011) Continuous rapid action value estimates. In: Asian conference on machine learning, pp 19–31

He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

Hsu Fh (1999) IBM's deep blue chess grandmaster chips. IEEE Micro 19(2):70–81

Kocsis L, Szepesvári C (2006) Bandit based Monte-Carlo planning. In: European conference on machine learning. Springer, Berlin, pp 282–293

Muthoo RBA (1996) A course in game theory by Martin J. Osborne; Ariel Rubinstein. Economica 63(249):164–165

Osborne MJ, Rubinstein A (1994) A course in game theory. MIT press

Silver D, Huang A, Maddison CJ, Guez A, Sifre L, Van Den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, et al (2016) Mastering the game of go with deep neural networks and tree search. Nature 529:484

Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T, et al (2017a) Mastering chess and shogi by self-play with a general reinforcement learning algorithm. Preprint. arXiv:171201815

Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A, Chen Y, Lillicrap T, Hui F, Sifre L, van den Driessche G, Graepel T, Hassabis D (2017b) Mastering the game of go without human knowledge. Nature 550(7676):354

Silver D, Hubert T, Schrittwieser J, Antonoglou I, Lai M, Guez A, Lanctot M, Sifre L, Kumaran D, Graepel T, et al (2018) A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science 362(6419):1140–1144