Dive into AlphaZero's Self-Play: Building Superhuman Intelligence from Scratch

Author: Tim Lin
Affiliation: DeepBioLab
Email: tim-lin@deepbiolab.ai

Introduction

Chess has long been regarded as a benchmark for artificial intelligence. Early researchers like Alan Turing and Claude Shannon laid the theoretical foundation for computer chess in the 1950s. Traditional chess engines such as Stockfish and Komodo rely on manually crafted evaluation functions, extensive opening books, and alpha-beta search algorithms refined over decades.

Similarly, Shogi and Go posed unique challenges to AI due to their complexity. Prior to AlphaGo, professional programs dominated these games:

- In chess, Stockfish analyzes about 70 million positions per second
- In Shogi, Elmo integrates extensive domain expertise

In 2016, researchers from DeepMind stunned the world by introducing an AI engine called AlphaGo. This AI defeated professional Go player Lee Sedol in a historic breakthrough. The complexity of Go far exceeds that of chess, with an immense number of possible board states, making professional-level AI engines previously thought unattainable.

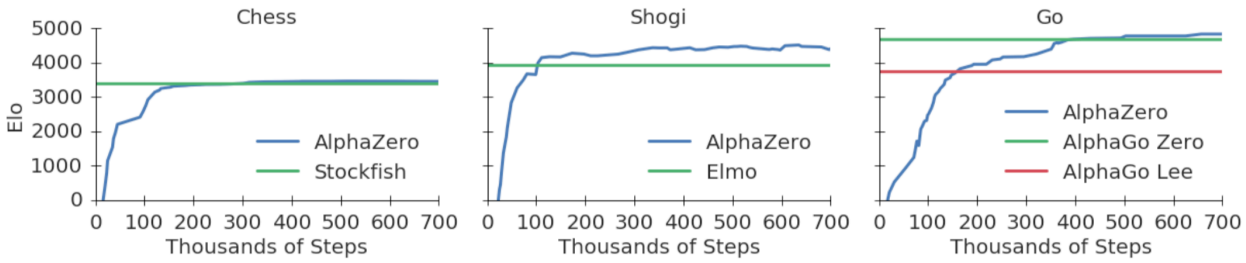
However, AlphaGo's training depended on expert game data, making it challenging to adapt to other domains.

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

David Silver,[1*] Thomas Hubert,[1*] Julian Schrittwieser,[1*]
Ioannis Antonoglou,[1] Matthew Lai,[1] Arthur Guez,[1] Marc Lanctot,[1]
Laurent Sifre,[1] Dharshan Kumaran,[1] Thore Graepel,[1]
Timothy Lillicrap,[1] Karen Simonyan,[1] Demis Hassabis[1]

[1]DeepMind, 6 Pancras Square, London N1C 4AG.
*These authors contributed equally to this work.

In 2017, the DeepMind team achieved another breakthrough by introducing AlphaGo Zero. This time, the algorithm completely abandoned expert data, learning the rules solely through self-play, and achieved even greater performance. Moreover, this algorithm was successfully applied to chess and Shogi (Japanese chess), demonstrating remarkable generality. Eventually, this new AI framework was named AlphaZero.



Comparison of Elo rating progress during AlphaZero's training with state-of-the-art programs. In chess (left), it surpassed Stockfish; in Shogi (middle), it surpassed Elmo; in Go (right), it outperformed previous versions of AlphaGo. All games began with random moves and provided only the game rules.

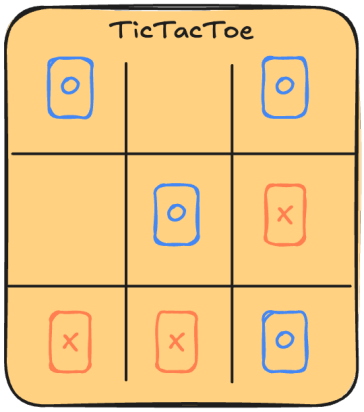
The revolutionary aspect of AlphaZero lies in its ability to learn entirely through self-play, starting with random moves and being provided only with the rules of each game. Unlike professional programs that rely heavily on extensive human knowledge and game-specific optimizations, AlphaZero demonstrated that a general learning algorithm could surpass these specialized systems while using significantly fewer computation resources per move.

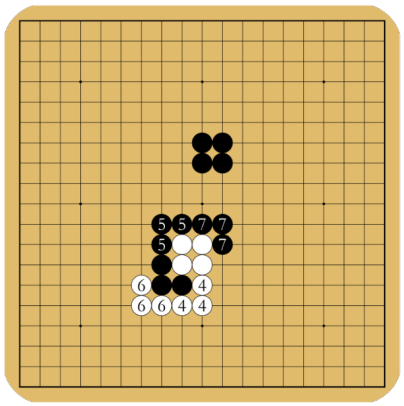
The biggest highlight of AlphaZero is its simplicity: it combines Monte Carlo Tree Search with deep neural networks. This approach closely resembles the thought process of human players—a perfect blend of calculation and intuition.

This article introduces the core principles and implementation methods of AlphaZero through visual diagrams. It focuses on analyzing its architecture that combines Monte Carlo Tree Search (MCTS) with deep neural networks, how self-play enables learning from scratch, and its applications and groundbreaking performance in chess, Shogi, and Go. Additionally, we will delve into AlphaZero's training process, strategy improvement mechanisms, and its advantages over traditional engines.

Zero-Sum Game

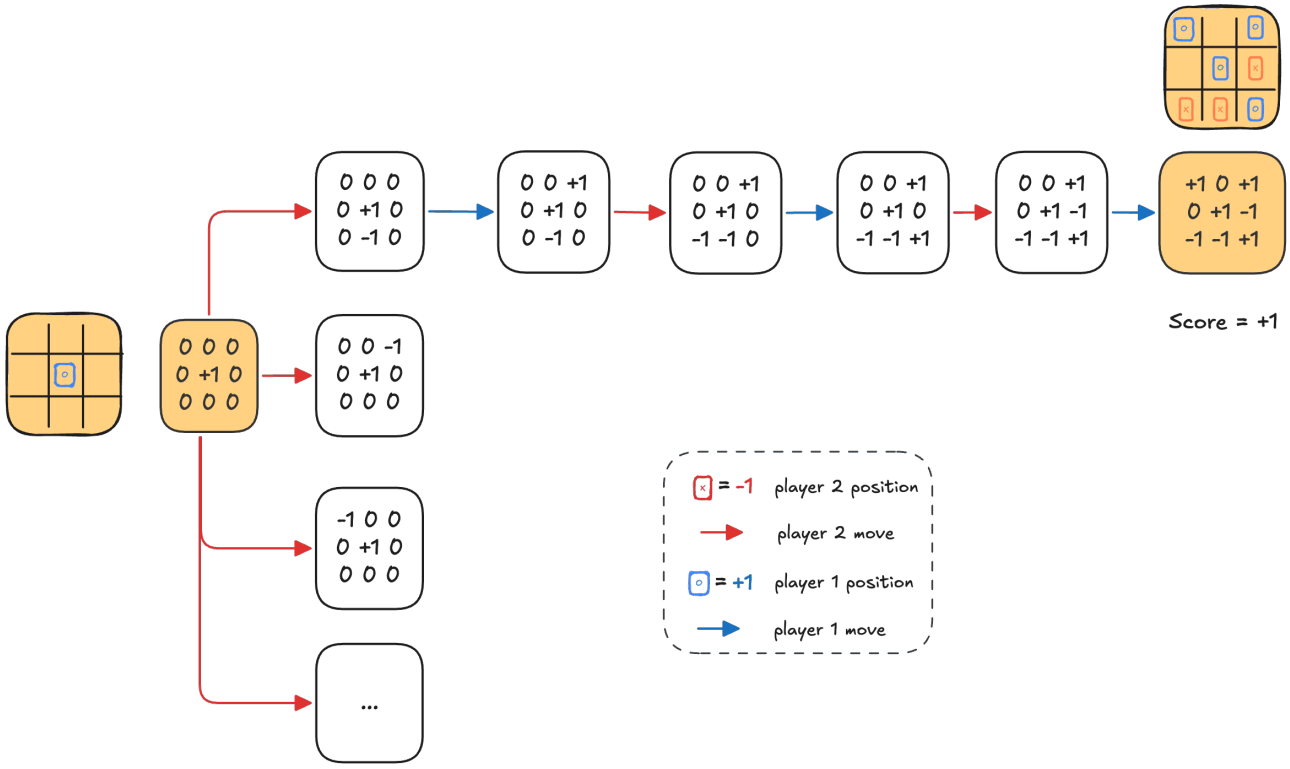
Before discussing AlphaZero, we need to understand the basic concept of a zero-sum game. Zero-sum games are an important theoretical foundation in artificial intelligence research, especially in competitive environments such as board games. The core characteristic of a zero-sum game is that **one player's victory necessarily means the other player's loss**. In such games, two players take turns making moves, with the goal of winning the match through strategic actions. Zero-sum games have the following characteristics:



1. **Opposing Outcomes**: The gain of one player equals the loss of the other. For example, in chess or Go, when one player wins, the other inevitably loses.
2. **No Hidden Information**: It is typically assumed that there is no hidden information or element of luck in the game. All information is public, and the outcome is entirely determined by the players' skills.
3. **Wide Applicability**: Zero-sum games are applicable to a wide range of games, from simple ones like Tic-Tac-Toe to more complex ones like chess and Go.

To better understand zero-sum games, let's take Tic-Tac-Toe as an example and represent it mathematically. Suppose the game state, i.e., the Tic-Tac-Toe board, can be represented as a matrix, where:



- `0` represents an empty space;
- `+1` represents Player 1's pieces (e.g., "O");
- `-1` represents Player 2's pieces (e.g., "X").

For example, the initial state of the board can be represented as a matrix filled with zeros. As players take turns, the values in the matrix are updated. The final game result can be expressed as a score:

- `+1` indicates Player 1 wins;
- `-1` indicates Player 2 wins;
- `0` indicates a draw.

This mathematical representation is highly convenient because it can be directly fed into neural networks and allows for simple matrix operations to handle player switching. For instance, multiplying the entire matrix by `-1` swaps the roles of the two players, and the score perspective can also be switched by multiplying it by `-1`.

Zero-Sum Games in Reinforcement Learning

By incorporating the above mathematical representation into the reinforcement learning framework, we can systematically describe zero-sum games. In reinforcement learning, each step of the game is considered a state, denoted as s_t. The two players take actions alternately:

- **Player 1 (denoted as +1)**: Acts on even-numbered steps with the goal of maximizing the final score z;
- **Player 2 (denoted as −1)**: Acts on odd-numbered steps with the goal of maximizing $-z$.

In this way, we can model zero-sum games as a dynamic programming problem, where each player aims to maximize their own payoff through an optimal strategy. Suppose we construct an agent capable of perfectly executing strategies as Player 1. By simply reversing the state matrix s_t and the score z, the same agent can also perfectly execute strategies as Player 2. This symmetry allows the agent to play against itself, continuously optimizing its strategy through repeated gameplay.

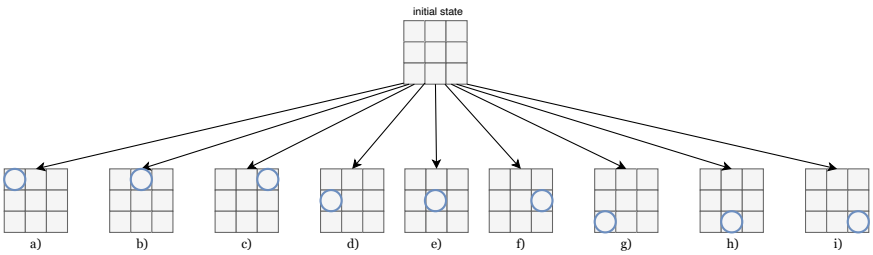
The core idea behind AlphaZero is precisely based on this symmetry in zero-sum games, enabling it to master optimal strategies for various complex board games (such as chess and Go) through self-play. In summary, AlphaZero includes two key components:

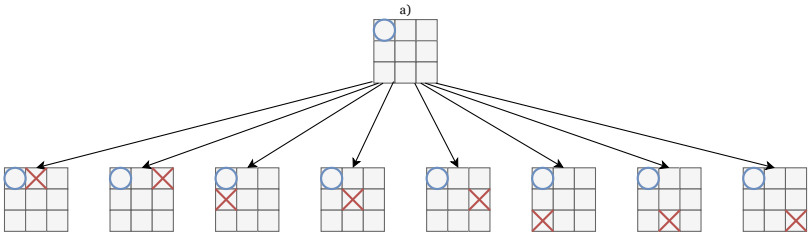
1. **Common Policy**: The agent uses the same policy regardless of whether it is acting as Player 1 or Player 2.
2. **Common Value Function**: A unified value function is used to evaluate the expected score from the perspective of the current player in the current state.

Monte Carlo Tree Search

How can we find an optimal strategy in a zero-sum game? In theory, this problem seems straightforward. For example, we could use brute-force search to explore all possible moves and all possible game outcomes, thereby selecting the optimal action path. However, in practice, this method is often impractical due to the sheer number of possibilities.

Take Tic-Tac-Toe as an example. Although it is a relatively simple game, the number of possible state combinations is still enormous.





Source: https://twice22.github.io/tictactoe/

In the first move, there are 9 possible choices, 8 in the second move, 7 in the third, and so on. This exponential growth makes brute-force search difficult to apply in practice. So, is there a more efficient way to solve this problem?

Initial Improvement Based on Random Sampling

Instead of traversing all possibilities to calculate the expected outcome, we can improve the approach by randomly sampling a subset of possible game paths:

- For each action, simulate multiple random games (playouts) and record the results of these simulations as $\{r_1, r_2, \ldots, r_n\}$, where r_i is the result of the i-th simulation (from the perspective of the current player).
- Compute the average result of these simulated games as the expected value of that action:

$$V(a) = -\frac{1}{n}\sum_{i=1}^{n} r_i \tag{1}$$

  A negative sign is added for convenience, so that the expected value is calculated from the perspective of the current player.

Next, select the action a^* with the highest expected value $V(a)$, i.e., find the action a^* that maximizes the expected value $V(a)$:

$$a^* = \arg\max_a V(a) \tag{2}$$

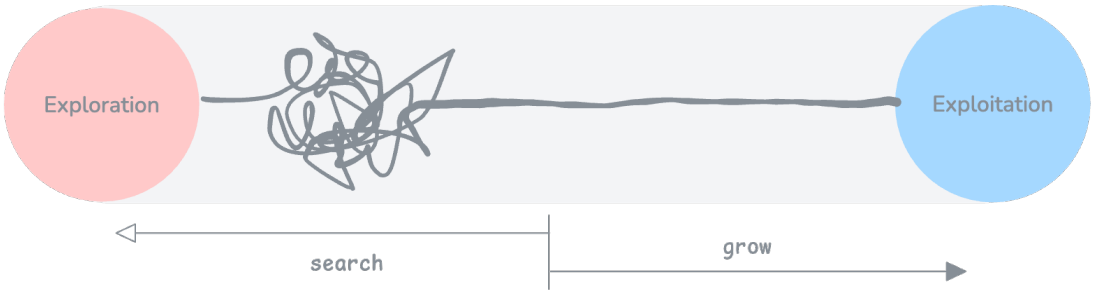
Systematic Exploration and Exploitation

The Exploration-Exploitation Tradeoff

This approach is similar to **Q-function estimation** in reinforcement learning. While random sampling is more efficient than brute-force search, purely random searches still have limitations. They lack guidance and cannot effectively balance exploration and exploitation:

- **Exploration** helps us gather new information and expand our knowledge boundaries, which is crucial for long-term decision-making.
- **Exploitation**, on the other hand, allows us to make optimal choices based on existing knowledge, leading to quicker and better results, especially when sufficient experience has been accumulated.

If we rely solely on known information (exploitation), we may miss better solutions or opportunities. Therefore, we need a more systematic method to address this tradeoff effectively.



Monte Carlo Tree Search

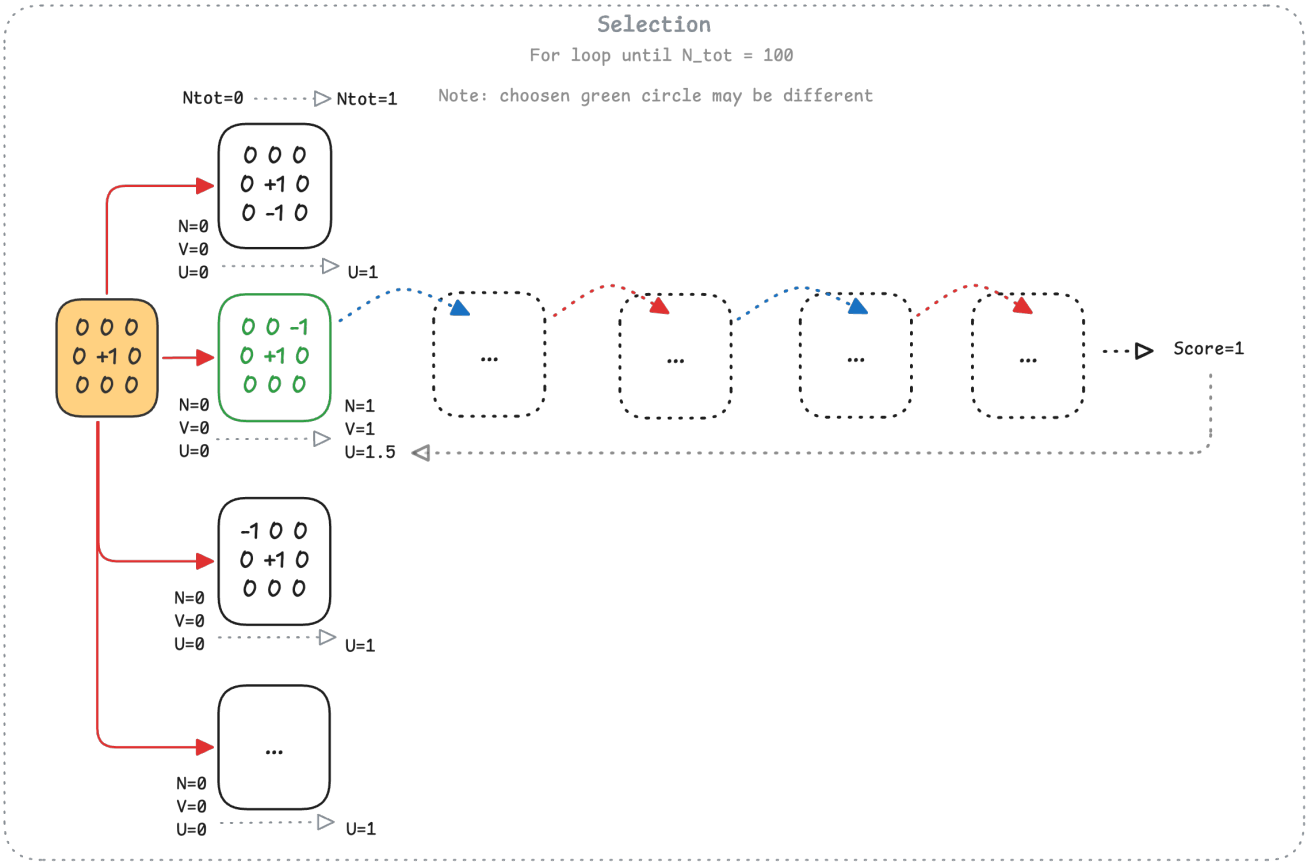
Monte Carlo Tree Search (MCTS) is an algorithm that combines random sampling with systematic search. It represents possible game states using a tree structure and dynamically adjusts resource allocation during the search, focusing more computation on promising branches.

In MCTS, each tree branch is associated with three key variables: **U**, **N**, and **V**.

- **N**: The number of random games (playouts) starting from this branch.
- **V**: The expected reward from this branch, calculated from the current player's perspective.
- **U**: A value that balances exploration and exploitation, used to decide which branch to explore next. The formula is as follows:

$$U = V + \frac{\sqrt{N_{\text{total}}}}{N+1} \tag{3}$$

Here, the first term V represents exploitation; the second term, inversely proportional to N, encourages exploration.



1. **Initialization**: Initialize U, N, and V for all branches to zero.
2. **Branch Selection**: During each iteration, select a branch based on its U value (as shown in the green box) for random game simulation.
3. **Value Update**: After completing the random game, update the corresponding branch's N and V values. Recalculate U for all branches (indicated by the gray dashed arrows).
4. **Repeat Iteration**: Repeat the above steps until reaching a predefined number of simulations (e.g., $N_{\text{total}} = 100$).
5. **Decision Output**: Finally, choose the branch with the highest visit count (N) as the next action. This is usually a more reliable choice since a higher visit count often indicates a branch with higher potential rewards.

After multiple iterations (N_{total}), each branch will have different visit counts and expected rewards. Ultimately, the next action is determined based on the branch with the highest visit count (N), rather than directly selecting the branch with the maximum U or V. This approach is more reliable because the visit count reflects the overall performance of a branch after extensive exploration.

The core advantage of MCTS lies in its ability to balance exploration and exploitation effectively:

- **Exploitation**: Prioritizes branches that currently appear to be the best, allowing for quick identification of potentially high-reward paths.
- **Exploration**: Simulates less-visited branches to avoid prematurely settling into a local optimum.

Enhancing Deep Search Capability

In the initial selection phase, MCTS chooses the best child node under the current node using the formula:

$$U = V + \frac{\sqrt{N_{\text{total}}}}{N+1} \tag{4}$$

However, this method is limited to exploring only one layer of the game tree. If we aim to predict longer action sequences, we need to delve deeper into the decision tree through **expansion** and **backpropagation** mechanisms.

- **Expansion**: When a node is visited, all its possible child nodes are generated, and their statistical variables U, N, and V are initialized.
- **Backpropagation**: After completing a simulation, the results are propagated step by step from the current node back to the root node. This includes updating the visit count N, the expected value V, and the total visit count N_{total}, thereby influencing the U values of the parent node and its sibling nodes.

Through expansion and backpropagation, MCTS can effectively explore deeper levels of the game tree, enabling more accurate evaluation of long-term strategies. After each simulation, the information for all relevant nodes is dynamically updated. This mechanism ensures that decisions are always made based on the most up-to-date data.

Expansion

The goal of the expansion phase is to explore new possibilities. When we select the best child node under the current node using the formula:

$$U = V + \frac{\sqrt{N_{\text{total}}}}{N+1} \tag{5}$$

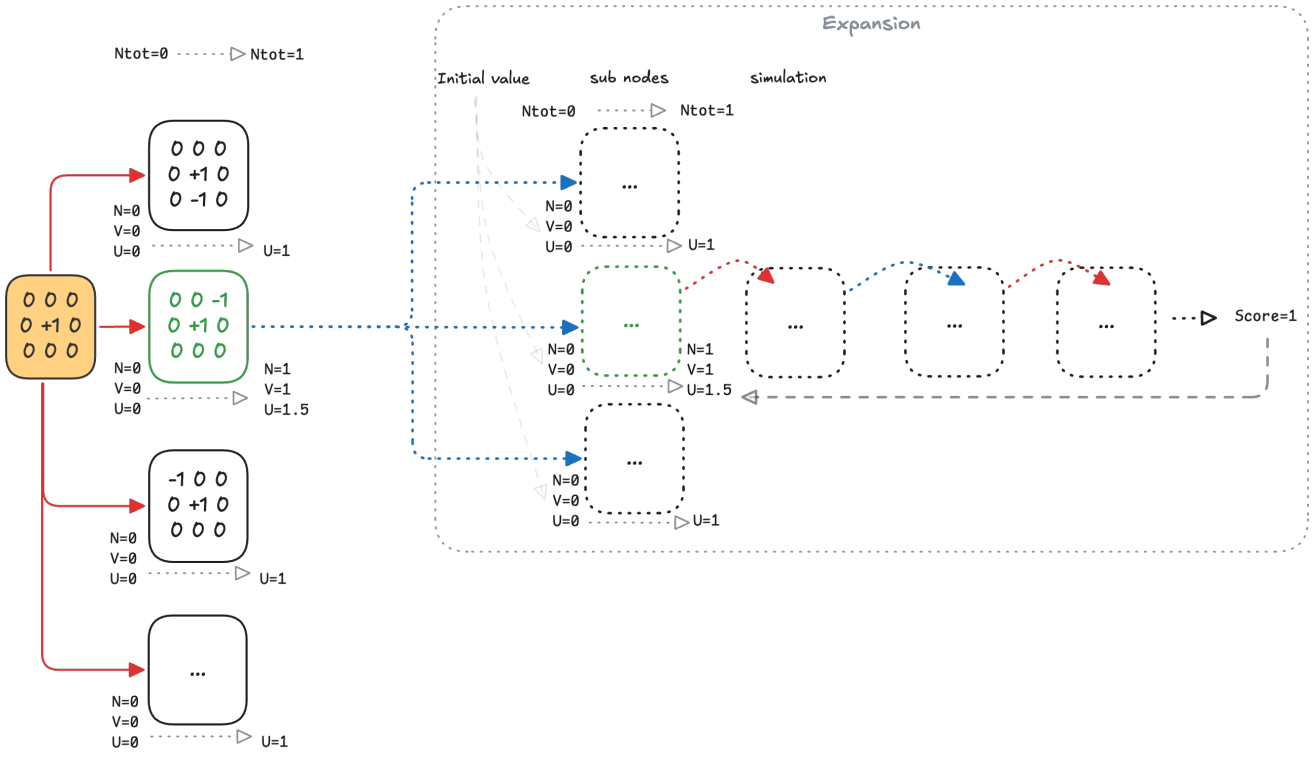
(as indicated by the green solid box), the following steps are performed based on this best child node:

1. **Generate Child Nodes**: List all possible next moves (blue dashed multi-branch arrows) and create a corresponding child node for each move.
2. **Initialize Variables**: Set initial values for each newly generated child node:

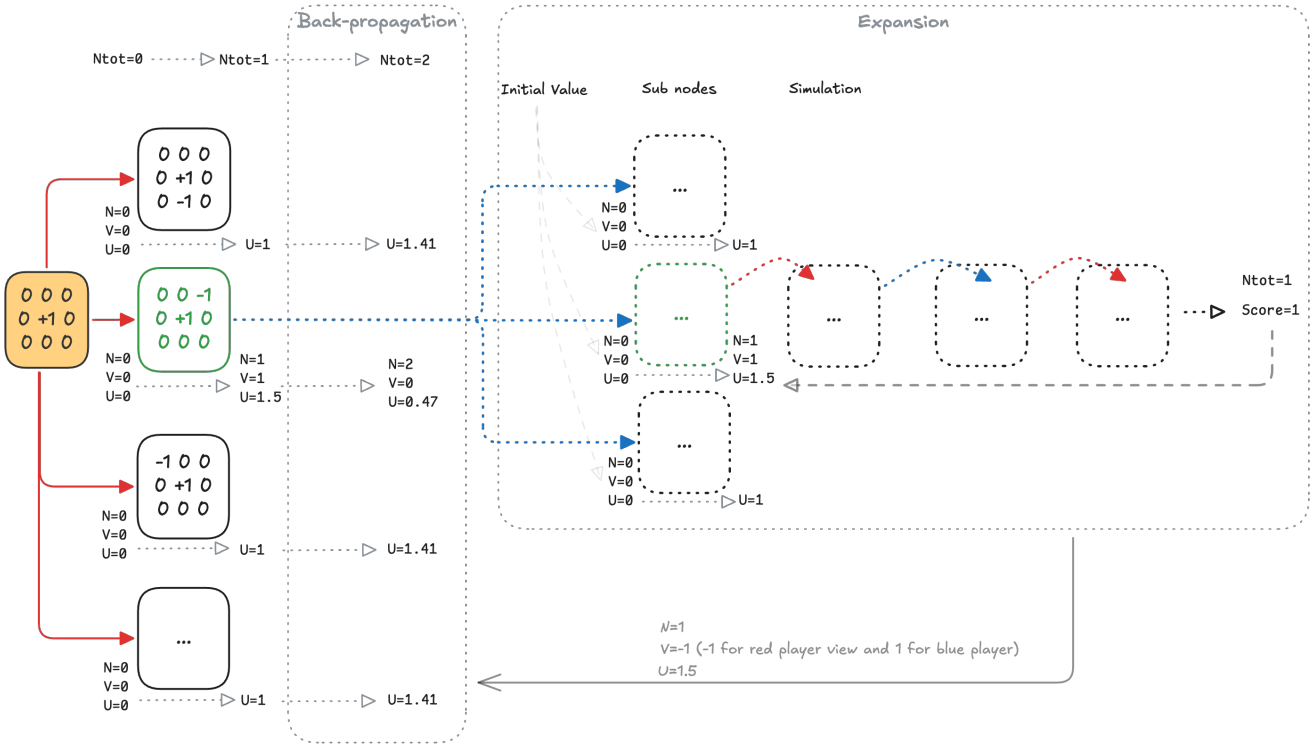
- $U = 0$: Initially unexplored, so the priority is zero.
- $N = 0$: Visit count is zero.
- $V = 0$: Expected value is yet to be determined.

3. Subsequently, a random action is selected from these child nodes (green dashed arrow) for simulation. The visit count N and expected value V of the randomly chosen action are updated. At the same time, the U values of other child nodes are recalculated using the updated total visit count N_{total}, according to the formula:

$$U = V + \frac{\sqrt{N_{\text{total}}}}{N + 1} \tag{6}$$



Backpropagation



After completing the simulation, the results need to be propagated step by step back to the root node. The specific steps are as follows:

1. **Update Visit Count** N: Increment the visit count of the current node and all its ancestor nodes by one.
2. **Update Expected Value** V: Recalculate the expected value of the current node using the formula:

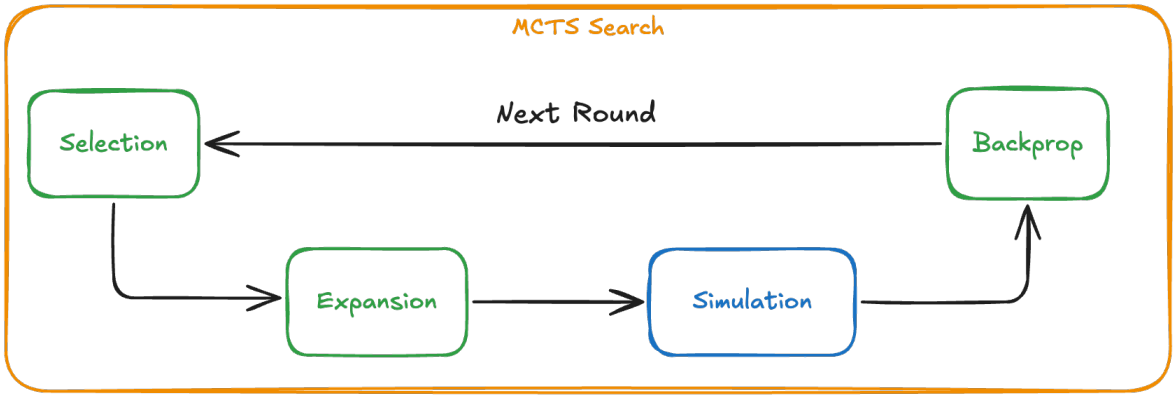
$$V = \frac{\text{Sum of all simulation results}}{\text{Visit count of the node}} = \frac{1 + (-1)}{2} = 0 \tag{7}$$

   Note that the expected value is calculated from the perspective of the current player.
3. **Update Total Visit Count** N_{total}: Update the total visit count for the entire subtree.
4. **Update Priority** U: Recalculate the priority of each node based on the new N, V, and N_{total} values.

Through backpropagation, the statistical information of the parent node and its sibling nodes is also adjusted, enabling more accurate guidance for future searches.

Monte Carlo Tree Search (MCTS)

By combining the three processes described above, we can construct the complete Monte Carlo Tree Search algorithm.



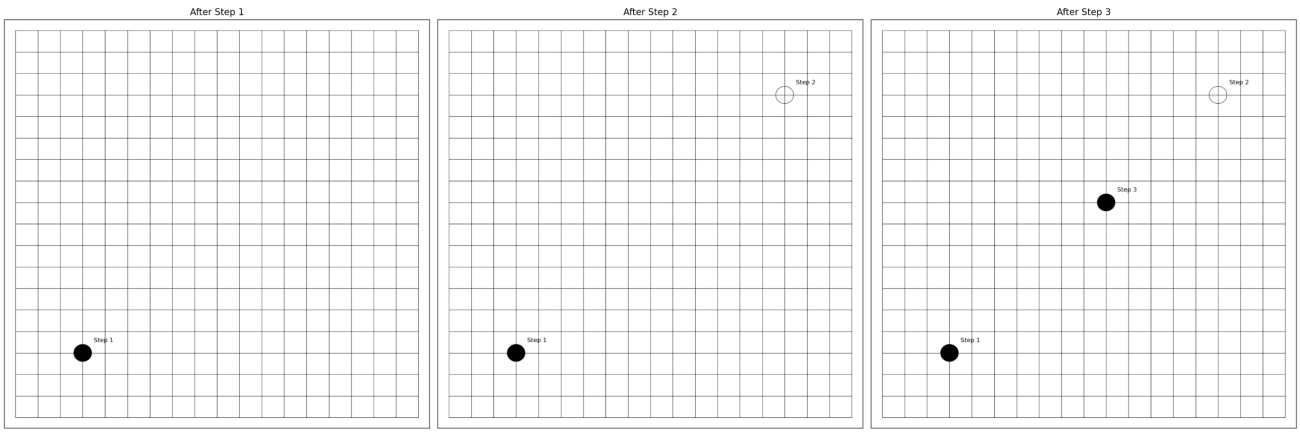
1. **Initialization**: Create a root node representing the current game state, and initialize its statistical variables.
2. **Iterative Search**: Repeat the following steps until the preset number of simulations is reached:
    - **Selection**: Start from the root node and traverse down the path with the highest priority until reaching a node that has not been visited or needs to be expanded.
    - **Expansion**: If an unvisited node is reached, generate all its child nodes.
    - **Simulation**: Perform a random game simulation starting from the current node until the game ends, and record the result.
    - **Backpropagation**: Propagate the simulation result step by step back to the root node, updating the relevant statistical information.
3. **Choose the Best Action**: Among all child nodes of the root node, select the one with the highest visit count (N) as the next action.

Monte Carlo Tree Search achieves efficient exploration of complex game trees by combining the expansion and backpropagation mechanisms. Its core idea is to use random simulations and statistical analysis to continuously optimize the decision path. As the number of simulations increases, the algorithm is able to explore deeper levels of the game tree, resulting in better decision-making.

Guided Tree Search

Challenges of Tree Search

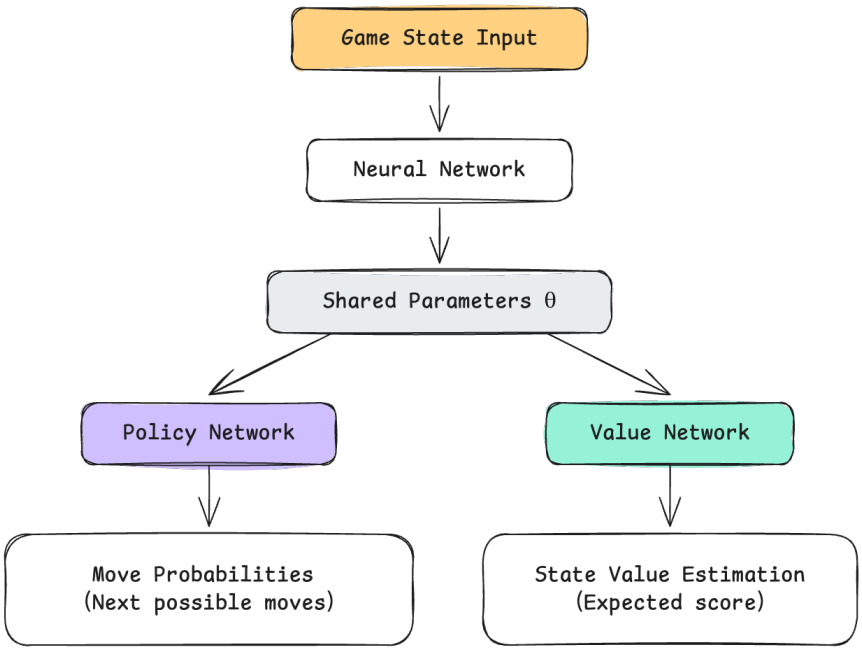
Tree search is an essential tool for solving decision-making problems, but its computational complexity grows rapidly with the scale of the problem. For example, in the game of Go, the board is a 19×19 grid, meaning there are 361 possible moves for the first step, 360 for the second, and 359 for the third. The combinations for just the first three moves already amount to approximately 50 million possibilities.



Although Monte Carlo Tree Search (MCTS) reduces some of the computational burden through random sampling, it is still difficult to accumulate sufficient statistical data to ensure reliable decision-making in such a vast search space. Therefore, a more efficient method is needed to improve MCTS.

Combine Neural Networks

AlphaZero improves traditional MCTS by incorporating neural networks. Specifically, it uses two key components:



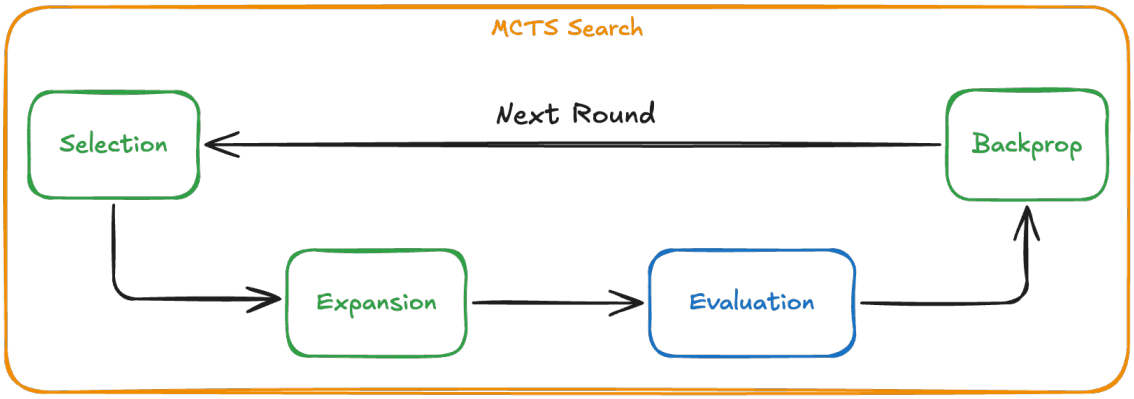
1. **Policy Network (π_θ)**: Used to predict the possible next moves of expert players.

2. **Value Network (v_θ)**: Used to evaluate the expected score for the current player starting from a given state.

These two components are implemented by the same neural network and share the same weights θ. The policy network helps MCTS focus on critical moves that experts are likely to choose, while the value network allows us to quickly estimate outcomes without fully simulating the game.

Improved Monte Carlo Tree Search

In AlphaZero, while the policy network can directly be used to select moves, AlphaZero combines it with MCTS to further optimize the strategy and find moves superior to the current policy.



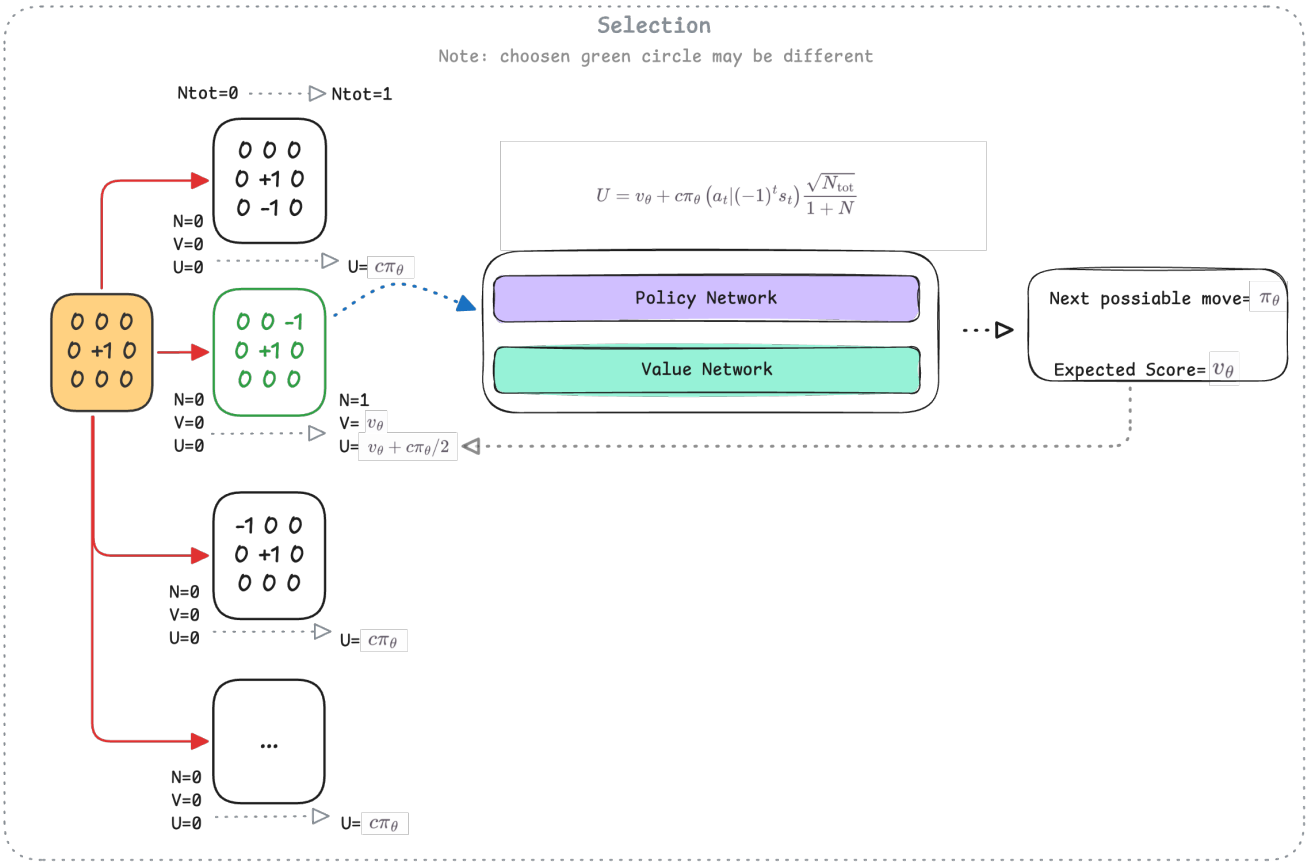
The following is the pseudocode and workflow for the improved algorithm:

```
def search(root):
    # Simulation steps
    for _ in range(num_simulations):
        node = root
        # Selection: Traverse the tree until reaching a leaf node
        while node.is_expanded():
            node = select_child(node)  # Use UCB formula with neural network priors

        # Expansion and Evaluation
        value = evaluate_position(node.state)  # Neural network evaluation
        backpropagate(node, value)

    # Return the improved policy
    return normalized_visit_counts(root)
```

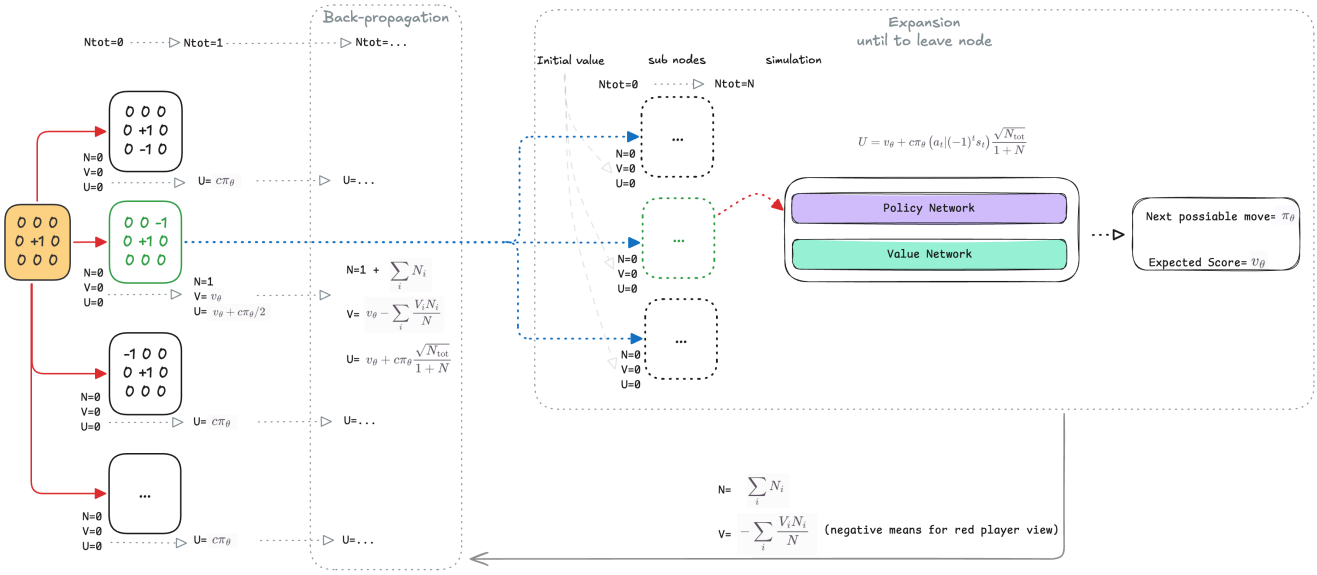
$$U = v_\theta + c\pi_\theta\left(a_t|(-1)^t s_t\right)\frac{\sqrt{N_{\text{tot}}}}{1+N} \tag{8}$$

1. **State Representation and Initialization**: For a given state, all possible actions are considered, and the following three values are assigned to each action:
    - **U**: The value used to guide the search.
    - **N**: The number of times an action has been visited.
    - **V**: The expected score.
2. **Calculation of U-Value**: The improved U function consists of two components:
    - The first part is the v_θ value, which controls exploitation of existing information.
    - The second part is an exploration term proportional to the probability output by the policy network π_θ, encouraging exploration of moves likely to be chosen by experts.
    - Additionally, a hyperparameter C is introduced to balance exploration and exploitation.

At the start of the search, all U values are initialized to zero, allowing a random action to be selected. Subsequently, the value network v_θ estimates the expected score for the current state, and the values of the relevant nodes are updated.



As the search progresses, when the same node is visited multiple times, it can be expanded into all possible subsequent states, and the U, N, and V values for each new state are initialized.



For non-terminal states, the V value is updated using a weighted average of the initial estimate and the exploration results from child nodes. Additionally, due to player alternation, a negative sign is added to the contributions of child nodes to ensure the estimation is from the perspective of the current player.

When the search reaches a terminal state, the value network is no longer relied upon, and the actual score is directly used to update the node values.

Selection of the Optimal Policy

After multiple iterations, the best action can be selected based on visit counts:

- To choose the **optimal policy**, simply select the action with the highest number of visits (N).
- To encourage more exploration, actions can be selected probabilistically based on their visit counts, using the formula $p_a = \frac{N_a}{N_{\text{tot}}}$.

Ultimately, these optimized moves are used to update the policy network, continuously improving the model's performance.

Self-Play Training

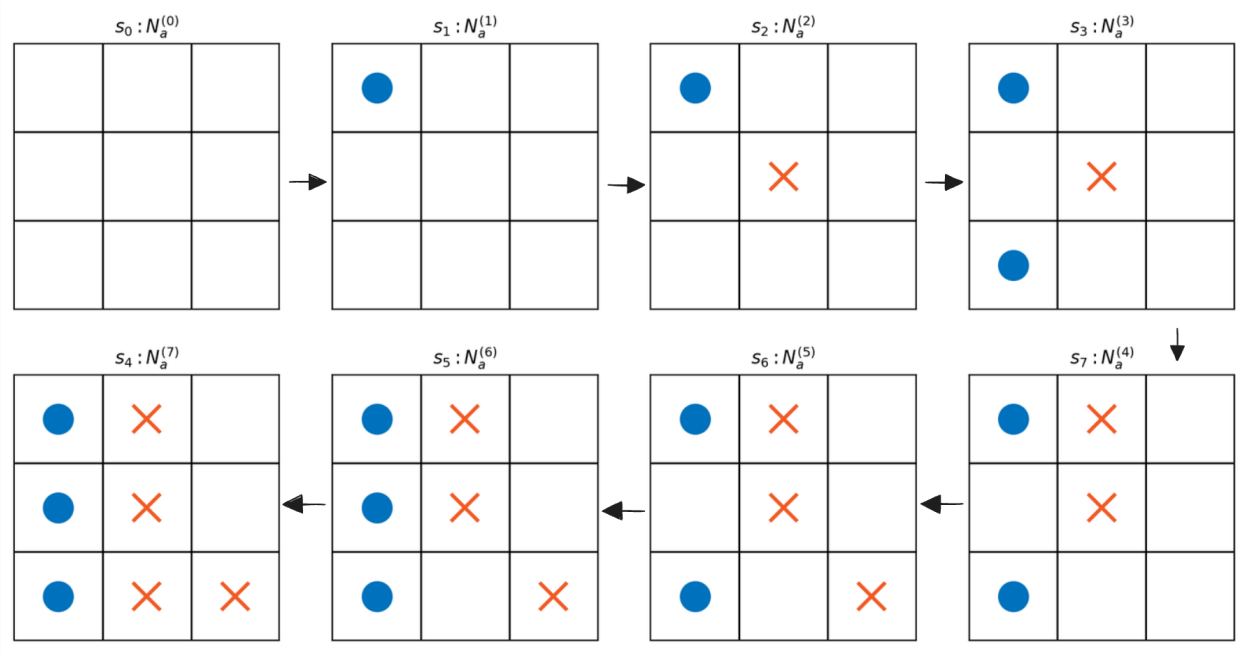
Monte Carlo Tree Search

AlphaZero's training begins with an empty board, such as in Tic-Tac-Toe. At this stage, the algorithm employs the current policy network (Policy) and value network (Critic) to perform Monte Carlo Tree Search.

At each step, MCTS generates a list of visit counts for the possible actions (N_a). These visit counts can then be converted into a probability distribution for each action:

$$p_a = \frac{N_a^{(t)}}{\sum_a N_a^{(t)}} \tag{9}$$

These probabilities reflect the tendency to select the optimal move in the current state.



After the player chooses the first move, the algorithm performs MCTS again from the new state. However, since certain states have already been explored, the algorithm does not need to start from scratch. Instead, it can effectively reuse previous search results. This incremental optimization makes MCTS more efficient in subsequent steps.

Loss Function

By repeating the above process, AlphaZero eventually reaches the game's final state. For example, in the Tic-Tac-Toe scenario mentioned earlier, the final outcome might be Player 1 winning ($z = +1$). Throughout the process, the expected results predicted by the value network can be compared to the actual final outcome to evaluate the accuracy of the value network.

$$L(\theta) = \sum_t \left\{ \left[v_\theta \left((-1)^t s_t \right) - (-1)^t z \right]^2 - \sum_a p_a^{(t)} \log \pi_\theta \left((-1)^t s_t \right) \right\} \tag{10}$$

- **Improves the Critic**: The part that optimizes v_θ improves the performance of the Critic model.
- **Improves the Policy**: The part that optimizes π_θ enhances the Policy.

These two components together form the loss function L_θ:

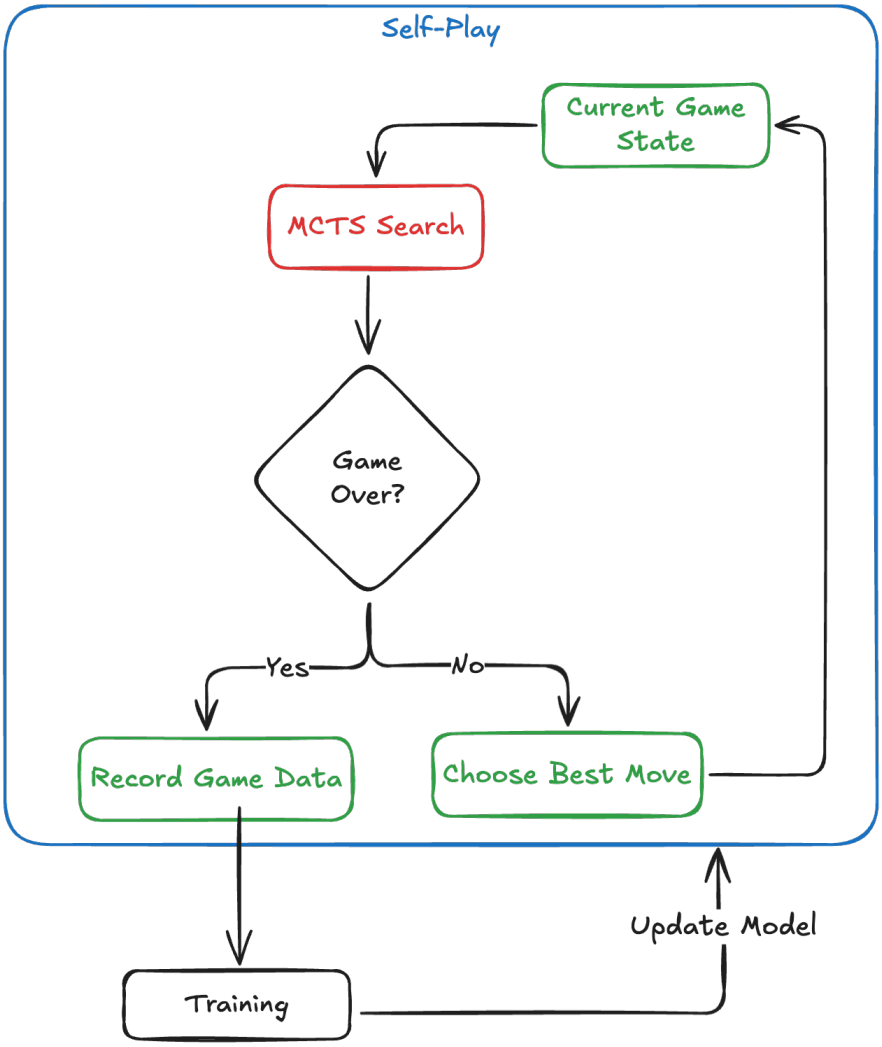
1. **Value Loss**: Measures the error between the value network's prediction and the actual final outcome. A factor of $(-1)^t$ is introduced to ensure that the comparison is always made from the perspective of the current player.
2. **Policy Loss**: By calculating the entropy difference between the policy network's predicted distribution and the MCTS-generated distribution, the policy network is forced to better align with the action choices suggested by MCTS.

$$\theta \leftarrow \theta - \alpha \nabla_\theta L(\theta) \tag{11}$$

By minimizing this loss function, AlphaZero can iteratively optimize its policy and value networks.

Training Loop

Once the loss function is defined, the algorithm can use gradient descent to update the policy and value networks. The core loop of the AlphaZero algorithm is as follows:



1. **Initialization**: Set up a neural network that includes both the policy and value networks.
2. **Self-Play**: Use the current networks and MCTS to select actions and complete a game.
3. **Network Update**: Compute the loss function and optimize the neural network using gradient descent.

By repeating the above steps, AlphaZero continuously improves its performance, evolving from a beginner to an expert.

Intuitive Understanding of the Learning Mechanism

Some might wonder: in the initial state, when the policy and value networks are completely random, how does AlphaZero achieve learning? The answer lies in the following key points:

1. **Learning from the Final Outcome**: Even in the early stages, when MCTS performs poorly, the value network can improve its prediction of final outcomes by propagating results through the tree search.
2. **Gradual Policy Improvement**: As the value network becomes more accurate, the policy network can also improve gradually, as it increasingly relies on the high-quality action probability distributions generated by MCTS.
3. **Layered Refinement**: In the early stages of training, AlphaZero tends to first master endgame strategies. As endgame performance improves, midgame strategies also improve correspondingly. Eventually, the algorithm can predict long sequences of expert-level moves, achieving comprehensive enhancement.

AlphaZero cleverly **integrates MCTS with neural networks**, **optimizing policy and value predictions step by step through feedback from final outcomes**, achieving self-improvement in an intuitive and efficient learning mechanism. This mechanism is similar to the process of reinforcement learning in the human brain, where decisions and value judgments are gradually optimized through feedback, strengthening neural connections. AlphaZero's self-play exemplifies autonomous learning and experience-driven intelligence emergence observed in cognitive neuroscience.

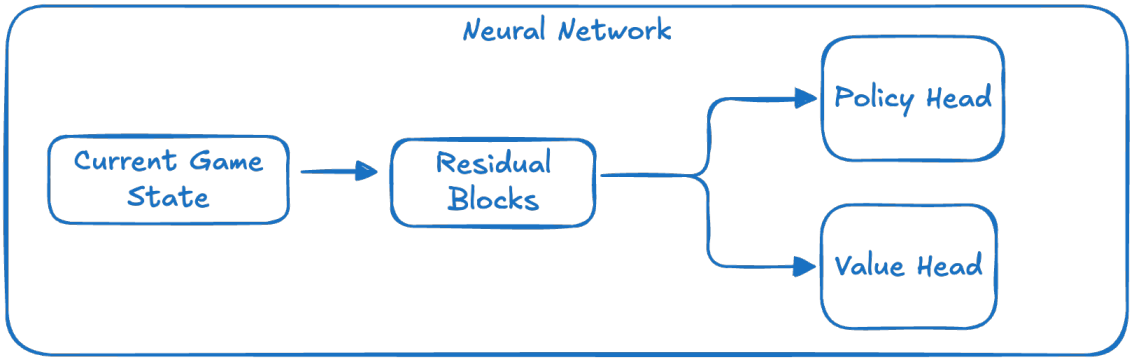
Innovations of AlphaZero

Returning to the AlphaZero paper, based on the explanations provided earlier, its core innovation is the integration of neural networks with Monte Carlo Tree Search (MCTS) within a self-play reinforcement learning framework. Compared to traditional programs, its key differences are:

- **No Human Knowledge**: AlphaZero starts without domain knowledge and only understands the rules of the game.
- **Neural Network Guidance**: AlphaZero uses neural networks to evaluate positions and suggest promising moves, rather than relying on extensive position evaluation.
- **Monte Carlo Tree Search**: AlphaZero employs MCTS, guided by neural networks, to efficiently explore the game tree.
- **Self-Play Training**: The system improves by repeatedly playing against itself, updating neural network parameters to predict game outcomes and optimal move choices.

Neural Network Architecture

AlphaZero uses the same neural network architecture for all three games, with only minor adjustments for different board sizes and move representations:



- **Input**: The game state is represented as a set of feature planes (including piece positions, repetition counts, etc.).
- **Processing**: Multiple residual blocks with convolutional layers.
- **Output**: Two heads— the policy head generates move probabilities, and the value head estimates the value of the position.

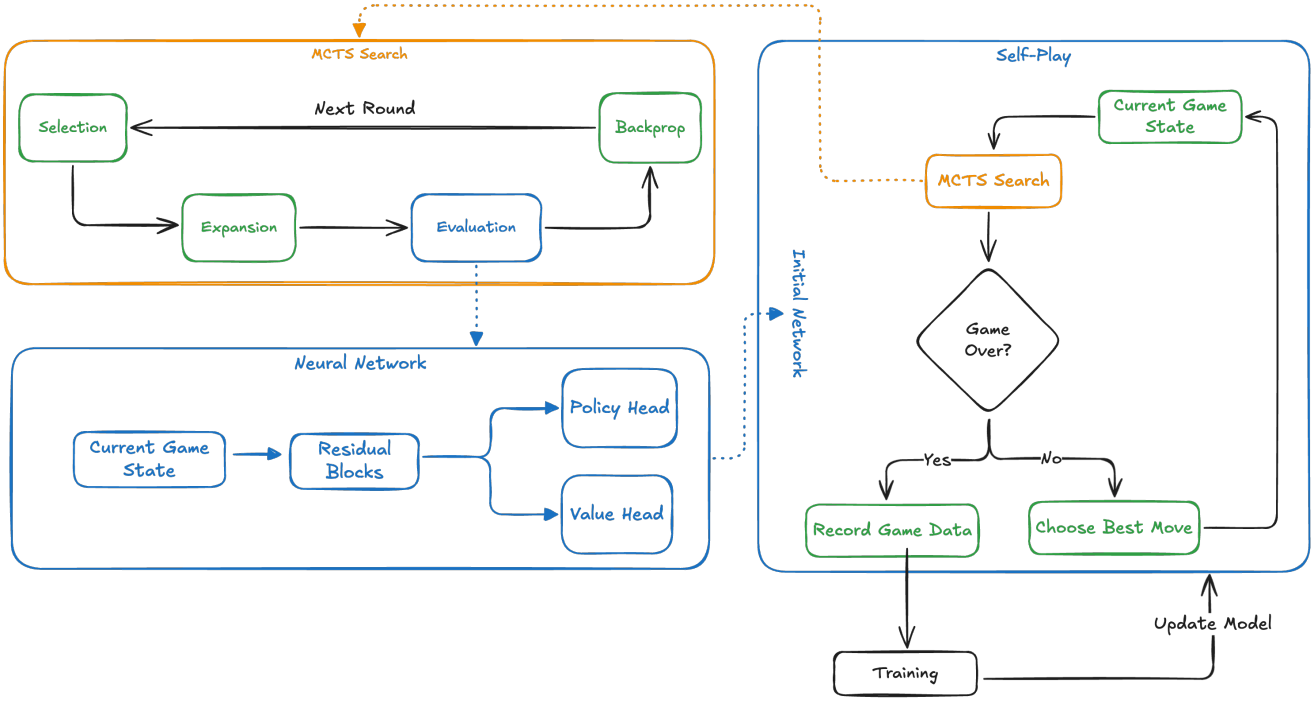
Mathematically, the network can be expressed as:

$$(p, v) = f_\theta(s) \tag{12}$$

Where:

- p is a probability distribution over moves.
- v is a scalar value between -1 and 1 (win/loss prediction).
- f_θ is a neural network with parameters θ.
- s is the current game state.

Self-Play Learning



The learning process of AlphaZero follows these key steps:

1. **Initialize the Neural Network**: The neural network is initialized with random weights.
2. **Self-Play Generation**: AlphaZero plays games against itself, using MCTS guided by the current neural network to select moves.
3. **Training Data Creation**: Each move in the self-play games creates a training example (state, improved policy, game outcome).
4. **Network Update**: The neural network parameters are updated to minimize the error between predicted and actual outcomes, and to maximize the similarity between predicted move probabilities and search probabilities.

The training objective function can be expressed as:

$$L = (z - v)^2 - \pi^T \log p + c||\theta||^2 \tag{13}$$

Where:

- z is the game outcome (-1, 0, or +1).
- v is the predicted value.
- π is the improved policy from MCTS.
- p is the policy predicted by the neural network.
- $c||\theta||^2$ is the regularization term.

This iterative improvement process continues for hundreds of thousands of steps, allowing the system to progressively discover stronger strategies without human input.

Results and Performance

AlphaZero achieved remarkable results across all three games:

- **In Chess**: After 4 hours of training, AlphaZero defeated Stockfish 8 with 28 wins, 72 draws, and 0 losses in 100 games.
- **In Shogi**: AlphaZero defeated Elmo with 90 wins, 2 draws, and 8 losses.
- **In Go**: After 8 hours of training, AlphaZero surpassed the performance of AlphaGo Zero.

These results are particularly impressive considering:

- AlphaZero searches approximately 80,000 positions per second in chess, compared to Stockfish's 70 million.
- AlphaZero uses the same general algorithm for all three games.
- AlphaZero does not rely on opening books or endgame databases.
- AlphaZero achieves superhuman performance in just 24 hours of training.

By leveraging neural networks, AlphaZero focuses its attention on the most promising options, demonstrating that the depth of computation can be effectively traded off against the breadth of learned knowledge.

Opening and Playstyle

Common Chess Openings Discovered by AlphaZero

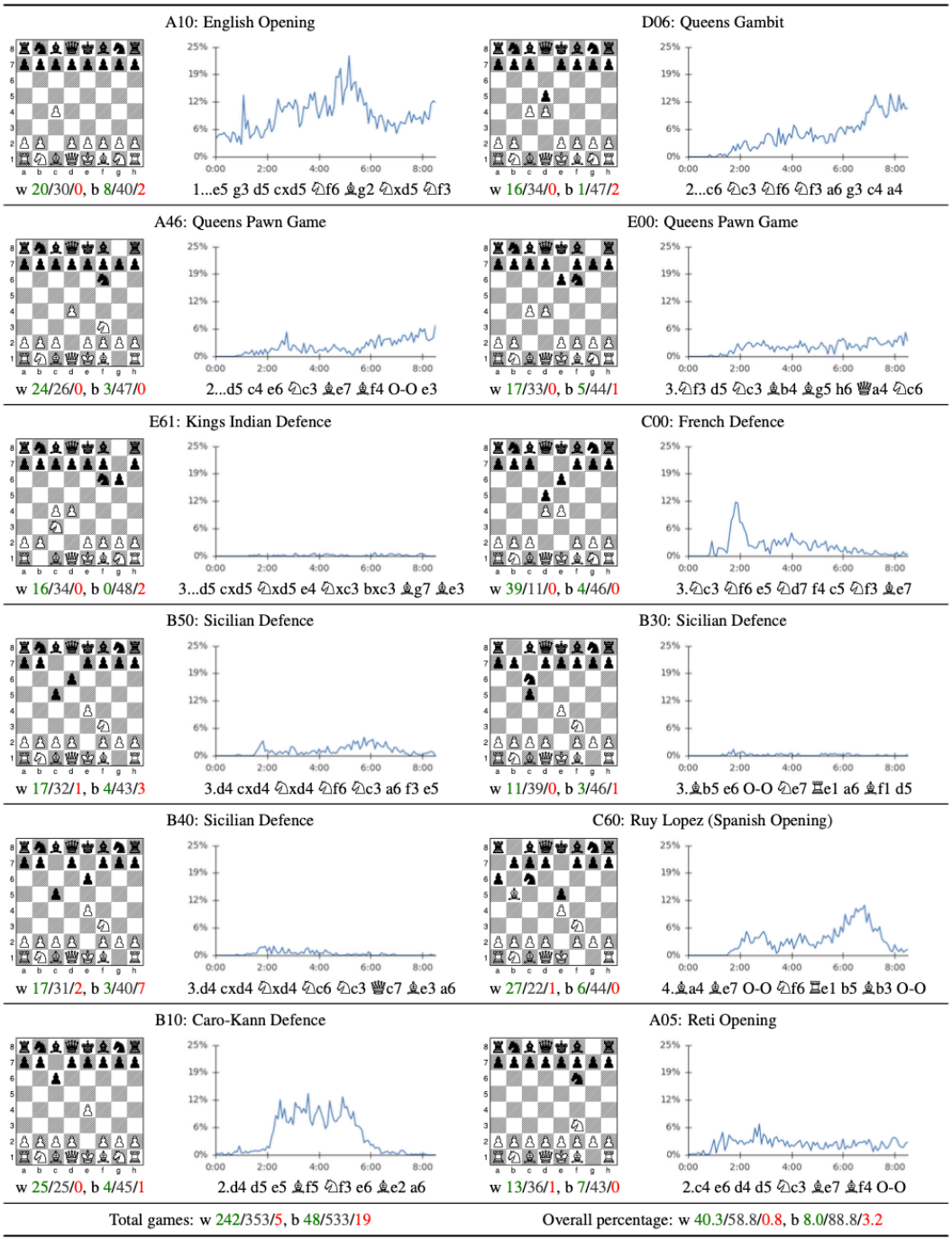


***Figure 1**: AlphaZero independently discovered and frequently used common human chess openings. The chart shows the frequency of these openings during different stages of AlphaZero's training.*

AlphaZero independently discovered mainstream human opening theories across games. In chess, it frequently used popular openings such as the English Opening, Queen's Gambit, and Ruy López. **Remarkably, AlphaZero discovered these openings through self-play rather than studying human games.**

Analysis of AlphaZero's games revealed its unique playstyle:

- **Dynamic Piece Play**: AlphaZero often sacrifices material for positional advantages and piece activity.
- **Long-Term Compensation**: It frequently makes moves that seem suboptimal in the short term but gain advantages later in the game.
- **Unconventional Methods**: It rejects many standard principles, such as traditional piece development rules, in favor of unique strategies.

AlphaZero demonstrated exceptional skill in open tactical positions, where its ability to accurately calculate complex variations gave it a significant advantage over traditional engines.

Scaling with Computation Time

Performance of AlphaZero vs. Stockfish and Elmo as Computation Time Increases

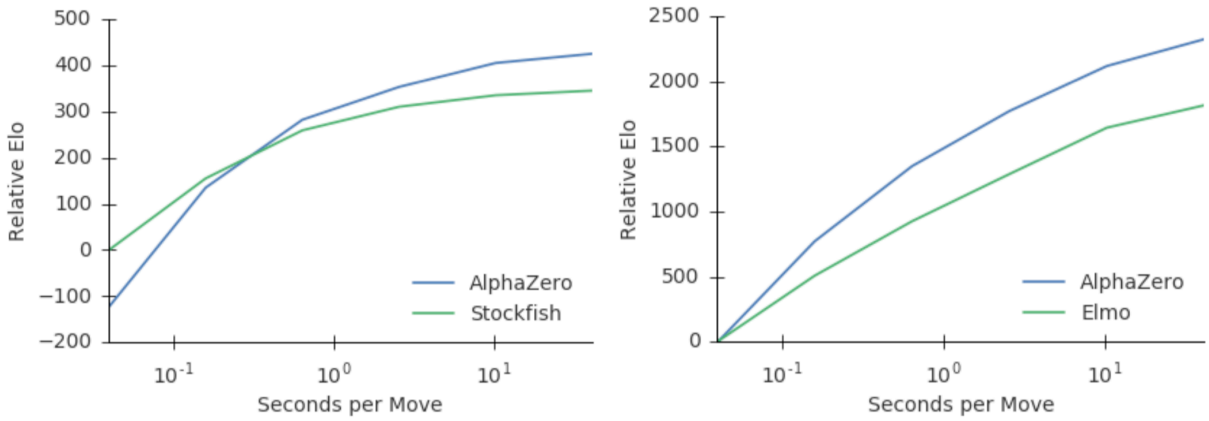


***Figure 2**: As computation time increases, AlphaZero's performance improves faster than that of Stockfish and Elmo. The left chart shows performance in chess, while the right chart shows performance in shogi.*

One of AlphaZero's key advantages is its **effective scalability with increased computation time**. While traditional engines like Stockfish and Elmo also improve with more computation time, AlphaZero's improvement curve is steeper.

As shown in Figure 2, AlphaZero initially underperforms Stockfish when computation time is very limited (less than 0.1 seconds per move in chess). However, as computation time increases, AlphaZero's relative Elo advantage grows significantly. This demonstrates that AlphaZero's MCTS approach, guided by learned knowledge, explores the game tree more efficiently than traditional alpha-beta search algorithms.

This scalability is particularly important for practical applications, where computation time may vary depending on the situation.

Impact and Future Directions

AlphaZero's success has profound implications for artificial intelligence research:

- **Generality of Reinforcement Learning**: The same algorithm achieving superhuman performance across multiple complex domains suggests that reinforcement learning can be widely applied to difficult problems.
- **Reduced Dependence on Human Knowledge**: The success of learning from scratch challenges the necessity of human expertise in complex domains.
- **New Approaches to Search Algorithms**: The effectiveness of neural network-guided MCTS challenges the dominance of traditional alpha-beta search in perfect-information games.
- **Potential Applications Beyond Games**: The general approach of combining neural networks with tree search can be applied to scientific discovery, optimization problems, and planning tasks.

Future research directions might include:

- Applying similar techniques to more complex games with hidden information.
- Extending the method to real-world problems with larger state spaces.
- Improving the efficiency of the learning process to reduce computational demands.
- Developing methods to extract and interpret the knowledge learned by the system.

AlphaZero is not just an achievement in game AI but a significant step toward more generalized artificial intelligence, capable of mastering complex domains without human guidance. By learning strategic principles from scratch through self-play, it demonstrates the potential for machine learning systems to independently discover knowledge, offering new insights in areas lacking human expertise or requiring novel approaches.

References

- David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, 和 Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587):484–489, January 2016.

- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, 和 Demis Hassabis. Mastering the game of go without human knowledge. Nature, 550:354–359, 2017.
- M. Campbell, A. J. Hoane, 和 F. Hsu. Deep Blue. Artificial Intelligence, 134:57–83, 2002.
- Claude E Shannon. XXII. Programming a computer for playing chess. The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, 41(314):256–275, 1950.