Alpha Zero Rotate 4

(Soon to be) Dr David Mills*
Department of AI
Utlandia
davidmills@GG.ut

Abstract

Ohhhh yeahhhh I did it again. I just watched the AlphaGo documentary, and oh my goodness, I am hyped. As part of the AI-fication of the company, I spent the entire night writing a Connect 4 environment and training a new, improved model, which I named AlphaZero. I think I had a bit too much gaming soda to drink during the development, because the game suddenly rotates (???). I'm not sure, but here is my paper on the discoveries and documentation of the work done. NeurIPs 2026 here I come :D. (PS: to any manager reading this, make room in the budget for 1st class tickets to America). I also created a tournament system for anyone willing to try to beat the machine.

1 Reinforcement Learning

Machine learning is typically split into three main categories. First comes supervised learning, the most well-known approach, training a network or model to map the input X to an output y through backpropagation. Secondly, there is unsupervised learning, where the input data has no clear label; therefore, one must find clusters and characteristics between the data entries in a dataset without knowing exactly what the label is. Lastly comes reinforcement learning.

Reinforcement learning is implemented such that an AI **agent** is placed in an **environment** that has a current **state**, and the agent can alter the state through doing an **action**. Whenever taking an action, the agent is rewarded with, you guessed it, a **reward**. The agent should then, based on the reward, alter its actions to get higher rewards, resulting in a smarter agent, which again picks better actions.

1.1 Environment

Ohh mighty the great environment. An environment can be everything from a game to the real world. Think of it as the universe where the learning happens. It must always have a current state (s), which is the complete snapshot of the world right now. When the agent takes an action (a), the environment reliably or randomly updates this state to a new state (s'). An environment can also change over time, but this is harder and outside the scope of this paper (I totally understand how it works).

1.1.1 Step

The **Step** function is the core physics engine. It takes the agent's action a and the current state s, and determines what happens next.

- **Input:** State s and Action a.
- **Output:** The new state s' and the reward r.

*Goood Games.

Preprint. Under review.

In short: the agent tries something, and the `Step` function calculates the consequence. If the environment is deterministic (like a Connect 4 game), the outcome is always the same. If it's stochastic (like Yahtzee), the outcome is a random roll of the dice!

1.1.2 Reward

The environment's honest, but brutal opinion. The **Reward** is the immediate, scalar feedback r the environment hands out after a step. It's the only thing the agent truly cares about; the agent's sole purpose in life is to maximise the **total future reward**.

- **Positive Reward (+):** "Good job! Have a cookie." (e.g., scoring a point).
- **Negative Reward (-):** "Nope, wrong move. That's a penalty." (e.g., crashing into a wall).

It is very much like training a dog. A well-designed reward function is essential. If you reward an agent for getting close to the goal but never reaching it, it might just orbit the goal forever, believing it's doing a fantastic job.

1.2 Agent

The **Agent** is the actor of an environment. It is the decision-making entity that observes the current state (s) and selects an action (a). Think of it as a mouse in a maze, where the mouse can die over and over until it finally learns the correct path, not falling into the lava.

At its core, the Agent uses its internal programming (its **policy**, π) to map the current state to the action it believes will maximize its long-term reward. The policy is the agent's ultimate cheat sheet for success.

1.2.1 Exploration and Exploitation

An agent learns best when balancing the **Exploration** and **Exploitation** of an environment. This is known as the Exploration-Exploitation Trade-off. It's the constant internal debate every good agent has:

- **Exploitation:** Taking the action currently known to yield the highest reward. This is like sticking to the one route you know leads to the cheese (playing it safe).
- **Exploration:** Taking a random, unknown, or less optimal action to discover new paths and potentially better rewards. This is like trying the spooky, dark corridor you've never used before (taking a risk for a bigger payoff).

A purely exploitative agent will quickly find a local maximum but might miss out on a globally better solution. A purely explorative agent will wander randomly forever, never truly optimizing its path. Neither alone is great, but balanced, it can achieve quite spectacular things.

How can this balance be achieved? The most common strategy is called the ϵ-greedy method.

- With a small probability ϵ (epsilon), the agent **explores** by choosing an action randomly.
- With the remaining probability $(1 - \epsilon)$, the agent **exploits** by choosing the best-known action.

Often, ϵ is started high and slowly reduced over time, allowing the agent to initially explore a lot and then settle down to exploit its knowledge.

1.3 Policy (π) and Value Function (V)

These two functions are the core of the agent's intelligence. The Policy is the agent's behavior—what it does. The Value Function is the agent's prediction—how good a situation is.

1.3.1 Policy (π)

The **Policy** (π) is the Agent's learned strategy. It is the direct rule that dictates the Agent's behavior by mapping the current state (s) to the action (a) it should take.

- **Deterministic Policy:** $\pi(s) = a$. Given a state, the agent always chooses one specific action. (e.g., "If I see a wall, turn left.")
- **Stochastic Policy:** $\pi(a|s)$. Given a state, the agent chooses actions based on a probability distribution. (e.g., "If I see a wall, I turn left 80% of the time and try to jump over it 20% of the time.")

The ultimate goal of training is to find the optimal policy (π^*), the one strategy that, if followed, will yield the greatest possible total expected reward.

1.3.2 Value Function (V)

The **Value Function** ($V(s)$ or $Q(s, a)$) is the agent's prediction of the total future reward it can expect to accumulate, starting from a given state (s) or taking a given action (a) in that state.

State-Value Function ($V(s)$): Answers the question: "If I start here and follow my current policy π, how much total reward will I get in the future?" This gives a score to a location. **Action-Value Function ($Q(s, a)$):** Answers the question: "If I am here (s), and I take this specific action (a), and then follow my policy π afterwards, how much total reward will I get?" This gives a score to a specific move.

The value function is the agent's internal scoreboard. Since the agent only wants to maximize reward, it uses the value function to evaluate its current policy: "Is this policy leading me to high-score states or low-score states?" If the action-value $Q(s, a)$ for one action is much higher than others, the agent should adjust its policy π to choose that high-value action more often!

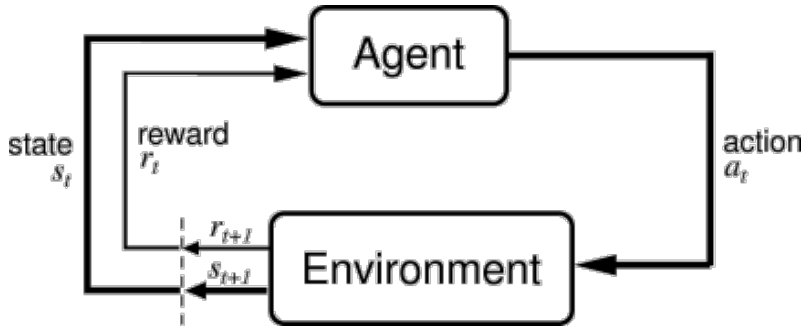

Figure 1: Agent and Environment. *(Source: https://www.researchgate.net/figure/Reinforcement-Learning-Agent-and-Environment_fig2_323867253)*

2 Rotate 4

Rotate 4 is a new spin on the original Connect 4 game. You play on a 7×7 board and can place a piece in each column. However, every 3–7 turns, the board rotates anticlockwise or clockwise randomly. This makes for a more interesting game of Connect 4, creating brand new strategies (Be aware I am currently awaiting a patent, so please don't make this into an app and become millionaires).

2.1 Connect 4

The base game remains the familiar classic, Connect 4.

- **Goal:** Get four of your pieces in a row (horizontally, vertically, or diagonally).
- **Action Space ($\mathcal{A}$):** The agent chooses one of the seven available columns ($A \in \{1, 2, \ldots, 7\}$, or 0-6 of you like 0 index) to drop a piece.
- **State Space ($\mathcal{S}$):** The state is primarily defined by the configuration of the pieces on the 7×7 grid.
- **Reward:** A massive positive reward (+2) is given for completing a 'Connect 4' (winning the game), and a large negative reward (-2) for the opponent completing one (losing the game). If it becomes a draw, the reward is +1 for the player who went second.

Crucially, in standard Connect 4, the Policy (π) and Value Function (V) only need to account for pieces staying exactly where they were placed. This simplicity is shattered by the rotation mechanic.

2.2 Rotation Aspect

The random rotation is the key differentiating feature, transforming the static game into a dynamic, non-stationary challenge. This has significant consequences for the Agent's learning process:

2.2.1 Impact on the Environment (State and Step)

- **State Space ($\mathcal{S}$):** The state must now explicitly include information about when the next rotation will occur, and perhaps how many rotations have already happened, as this is critical to predicting future board configurations.
- **Step Function:** The `Step` function now has two phases:
  1. The agent's piece drop and gravity calculation.
  2. Randomly (3–7 turns): An extra action is applied where the entire board state is mathematically transformed by a 90° clockwise or anti-clockwise rotation. All pieces move, making the board configuration s' very different from s even without the agent's piece drop.

2.2.2 Impact on the Agent (Policy and Value)

- **Local vs. Global:** The agent cannot simply evaluate moves based on immediate threats or gains, because a perfect three-in-a-row could be completely broken or magically completed on the next rotation.
- **The New Optimal Policy (π^*):** The optimal strategy must involve placing pieces that are rotation-proof (still beneficial regardless of rotation) or placing pieces that leverage the rotation to complete a line that would otherwise be impossible.
- **Value Function (V):** The Value Function $V(s)$ becomes much harder to estimate. The agent must predict the value of a state based on its current configuration and the probability distribution of all possible future rotated states it could become. It's a complex prediction problem!

This randomness forces the Agent to think much further ahead and develop robust, generalized strategies rather than relying on memorized sequences.

3 Monte Carlo Tree Search

For highly complex decision-making scenarios, such as games (e.g., Chess, Go) or sophisticated logistics problems, the number of possible outcomes can be astronomically large, often referred to as a high branching factor. In these cases, it is computationally impossible to determine the absolute best move by testing every single possibility.

Instead, one must rely on approximations and simulations. Here comes the Monte Carlo Tree Search (MCTS), which is an efficient, heuristic search algorithm used to find the best moves by intelligently building a search tree. It focuses computational effort on the most promising moves through a process of repeated simulations.

3.1 The Four Core Stages of MCTS

The MCTS process involves four core, iterative stages: Selection, Expansion, Rollout (Simulation), and Backpropagation. The stages are illustrated in Figure 2.

3.1.1 Selection

The selection stage is crucial for balancing Exploitation (choosing moves that have performed well historically) and Exploration (choosing moves that haven't been tried much, to ensure no hidden good moves are missed).

Starting from the current state (the Root Node), the algorithm traverses the existing tree by repeatedly picking the child node that maximizes the Upper Confidence Bound 1 (UCB1) formula. This formula provides an exploration bonus to nodes that have been visited less often. The algorithm continues selecting nodes until it reaches a node that is not yet fully explored (a Leaf Node).

The UCB1 formula for selecting a child node i from its parent node p is:

$$\text{UCB1} = \bar{X}_i + C\sqrt{\frac{\ln N_p}{N_i}}$$

Where:

- $\bar{X}_i$: The Average Reward (exploitation term) of the child node i. This is calculated as the total accumulated reward divided by the number of visits (W_i/N_i).
- N_i: The Visit Count (how many times we've tried this move) for node i.
- N_p: The Visit Count of the parent node p.
- C: The Exploration Parameter, a constant that determines the strength of the exploration bonus. The term $\sqrt{\frac{\ln N_p}{N_i}}$ is the Exploration Term.

3.1.2 Expansion

When the selection process reaches a Leaf Node (a node whose outcomes have not all been explored), the algorithm performs the expansion. This stage is simple: one new child node is created for one of the unvisited, legal actions available from the current state. This new node represents a move that is about to be evaluated for the first time.

3.1.3 Rollout (Simulation)

A rollout is a simulation performed starting from the newly created node. Since the value of this new state is unknown, the simulation proceeds by picking random actions for both sides continually until a terminal state (e.g., game over, end of a process) is reached. The outcome of this random playthrough is recorded as the Reward. This is the core "Monte Carlo" sampling aspect of the algorithm.

3.1.4 Backpropagation

After a rollout is complete and the terminal state reward is determined, this reward is backpropagated (passed back up) through the tree. Every node along the path from the newly created node back to the original root node has its statistics updated:

- The Visit Count (N_i) is incremented by 1.
- The total Reward (W_i) is updated with the outcome of the simulation.

This ensures the stored values accurately reflect the simulation results, making the selection stage smarter in the next iteration.

3.2 Nodes

Each node of the tree represents a specific state (e.g., a board position in a game) and holds the statistics critical for the UCB1 calculation: the Visit Count (N_i) and the Total Reward (W_i) gathered from all simulations that passed through this state.

3.3 Final Selection

After running the MCTS process for a predetermined number of iterations (n) or a set time limit, the algorithm must choose the final move. The best move to take from the current root state is typically the child node with either the highest visit count (N_i) or the highest average reward ($\bar{X}_i$). The move with the highest visit count is often chosen because it signifies the move that the sophisticated UCB1 formula consistently deemed the most promising over thousands of simulations.

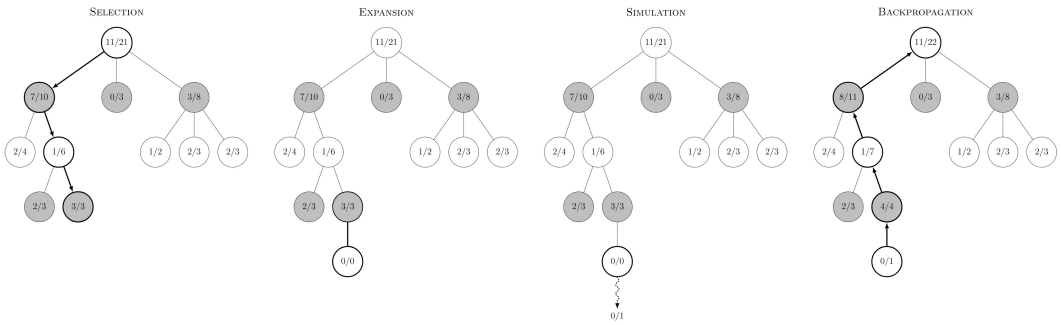

Figure 2: MCTS stages. *(Source: `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`)*

4 AlphaZero Implementation in the Rotate 4 Environment

This section explains the author's approach to implementing the AlphaZero algorithm for the Rotate 4 environment. AlphaZero is a machine learning agent that uses a Deep Neural Network (DNN) to dramatically enhance the efficiency and power of the Monte Carlo Tree Search (MCTS) algorithm. *Note that the company has yet to find the funding for deploying this monstrosity of an AI agent. Yet again, the author's brilliance is limited by politics and the so-called "lack of funding."*

4.1 Monte Carlo Tree Search (MCTS)

In this implementation, the core decision-making structure remains MCTS, but it is executed using information provided by the Neural Network instead of random simulations. The traditional four stages of MCTS are modified:

- **Selection:** Uses the PUCT (Polynomial Upper Confidence Trees) formula, a variation of UCB1, which incorporates the Policy Header's probabilities to intelligently guide the search toward moves the network predicts are most promising.
- **Expansion:** When a new node is created, the current game state is immediately processed by the Neural Network. The policy probabilities and value prediction are stored at the node, replacing the need for an initial random simulation.
- **Rollout Elimination:** Random rollouts are completely eliminated. The final value prediction from the Value Header of the Neural Network is used as the node's immediate, informed estimate of the position's worth, making the search significantly faster and more targeted.
- **Backpropagation:** The predicted value from the network is passed up the tree. All visited nodes along the path update their Visit Count (N) and their Total Value (W), thereby refining the statistical foundation for the next MCTS run.

4.2 Policy Header

The Policy Header is the part of the Neural Network responsible for telling the MCTS where to look.

- **Function:** It takes the current game state as input and outputs a probability distribution (P) over all possible legal actions. It assigns a likelihood to every possible move being the best first move from that state.
- **Impact on MCTS:** These probabilities are used in the Expansion phase to initialize new nodes. This gives high-probability moves a preference in the PUCT formula, ensuring the MCTS focuses its exploration on the moves the trained network deems most valuable.

4.3 Value Header

The Value Header is the part of the Neural Network responsible for telling the MCTS how good the current position is.

- **Function:** It takes the current game state as input and outputs a single scalar value, v, ranging from -1 to $+1$.
- **Interpretation:** $v = +1$ indicates the network predicts a certain win for the current player; $v = -1$ indicates a certain loss; and $v \approx 0$ indicates a draw or undecided outcome.
- **Impact on MCTS:** This value is used directly in the Rollout Elimination step. By using this informed prediction, the AlphaZero MCTS can evaluate the worth of a position much more accurately and rapidly than standard MCTS, which relies on hundreds of random game outcomes.

4.4 Training Cache

The Training Cache, often referred to as a Replay Buffer, is a critical component for the stability and efficiency of the Neural Network training process.

- **Purpose:** The cache stores the training data generated during the **Self-Play** phase. This data consists of triplets:
$$\langle S_t, \pi_t, z_t \rangle$$
  - S_t: The game state (board position) at time t.
  - π_t: The improved move probabilities derived from the MCTS search at state S_t.
  - z_t: The final game outcome (win, loss, or draw) from the perspective of the player at state S_t.
- **Training Benefit:** During the network training cycles, the network is trained on a random sample (a batch) of data drawn from this cache, rather than just the most recent games. This ensures the network trains on a diverse range of positions and outcomes, preventing it from overfitting to recent gameplay and leading to more stable and robust learning.

5 Tournament Implementation and Assessment

This section explains the structure of the automated tournament used to evaluate the performance of submitted player agents.

5.1 Rules of Engagement

The primary objective of the tournament is to objectively assess the strategic competence of a player's submitted agent against a fixed set of opponent bots.

- **Opponent Set:** Each player's agent must play against all available opponent bots in the system.
- **Games Per Matchup:** For each unique opponent bot, the player's agent will play a fixed total of **50 games**. This large sample size ensures statistical significance and reduces the impact of luck.
- **Assessment Metric:** The agent's performance is determined by its win rate across all games played. This win rate determines if the agent is deemed sufficiently competent to receive the flag.

5.2 Opponent Agents

Each opponent is implemented by inheriting from the main class `Agent`. They all support the key function `get_action(environment: Rotate4Env)`, which takes the current game state and returns the chosen move. The agents are ordered by increasing difficulty.

5.2.1 Random Agent

The `Random Agent` represents the baseline, minimal competence. It simply picks a random legal action available in the current game state, with no strategy or "thinking" whatsoever. This bot should be **trivial to beat**, as any agent demonstrating strategic thought should perform significantly better than pure chance. An agent performing worse than this is not considered a viable solution.

5.2.2 MCTS Agent

The `MCTS Agent` is a series of bots implemented using the Monte Carlo Tree Search algorithm. Its strength varies significantly based on a single parameter: the number of iterations (simulations) performed per action.

The agents that will be present during the tournament are shown in the table below:

Table 1: List of Agents and their iterations per move that will be present in the tournament.

Agent	Iterations per move
Random	0
MCTS	10
MCTS	50
MCTS	100
MCTS	200
MCTS	300
MCTS	500

5.2.3 Alpha Zero

The `Alpha Zero` agent represents the pinnacle of opponent strength. It is built upon a sophisticated version of MCTS that incorporates a Deep Neural Network. This network is used to inform the search:

- **Policy Head:** Determines which moves should be explored during the rollout, replacing the random choice with a trained probability distribution.
- **Value Head:** Determines the estimated value of a node, allowing the system to cut short unprofitable simulations.

Having been implemented through extensive training over thousands of hours, the Alpha Zero agent should be **unbeatable on paper** for all but the most advanced and carefully tuned player agents. *If the company had some more funding for a couple of A100 GPUs, the model could be deployed (The author has already trained it). This is sadly not the case, as some departments spent a yearly budget in a single month on pizza alone...*

5.3 Submission and Assessment via Client Handout

The tournament is run using a client-server architecture. Players do not upload their agent code directly but instead use the provided `client.py` handout to connect their local agent instance to the remote tournament server.

1. **Local Agent Implementation:** Players must implement their custom logic within their local `Agent` class.
2. **Client Connection:** The player executes the `client.py` program locally. This client initiates a connection to the central tournament server.
3. **Game Exchange (API):** The server manages the games against the various opponent bots. The server communicates the current game state (the `environment`) to the player's local client. The client, in turn, calls the player's local `Agent.get_action()` function and sends the resulting move back to the server.
4. **Tournament Run:** The server handles the entire 50-game match against each opponent sequentially.
5. **Score Reporting:** Once all required games are completed, the server calculates the total winrate, and if over 80%, it awards the client with a flag.

This client-server model allows the tournament to run securely and ensures all agents are assessed under identical network and environmental conditions.