Training RL Agents in Super Tic-Tac-Toe: PPO and AlphaZero Approaches

DONG Yifei(21114529)
Big Data Technology
Computer Science and Engineering
ydongbl@connect.ust.hk

ZHENG Mingen(21126390)
Big Data Technology
Computer Science and Engineering
mzhengap@connect.ust.hk

Abstract

This experiment investigates reinforcement learning techniques for playing Super TicTac-Toe on a cross-shaped board. We implemented two distinct approaches for TicTacToe game: a Proximal Policy Optimization (PPO) algorithm using TorchRL for the 12×12-cross-shaped variant, and an AlphaZero implementation for standard 3×3 TicTacToe. The results showed the effectiveness of our implementation. Our code can be reached at https://github.com/Captain-Named/RL25HM2.

1 PPO with TorchRL for Super TicTacToe

We implemented a Reinforcement Learning agent based on PPO (Proximal Policy Optimization) for the cross-shaped Super TicTacToe game where players take turns placing pieces in the cross-shaped area, winning by forming 4 consecutive pieces horizontally or vertically, or 5 consecutive pieces diagonally.

In our approach, we employed deep convolutional neural networks as policy and value networks and designed reward functions to guide the agent in learning effective game strategies. Through TorchRL's data collectors, experience replay buffers, and PPO objective functions, we built a complete training pipeline that enables the agent to continuously improve its performance through self-play.

1.1 Environment

We implemented a custom environment for our Super TicTacToe game by extending TorchRL's EnvBase class. This approach allowed us to integrate seamlessly with TorchRL's ecosystem while defining our specific game mechanics. We customized the environment's specification using Composite to define the observation space (a 2×12×12 tensor representing the board state), action space (an 80-dimensional categorical space for possible moves), and additional components like action masks and step counters. This structured representation enables the agent to understand the game state and make valid moves within the cross-shaped board.

1.1.1 Reset Method Implementation

The `_reset()` method initializes or resets the game environment to its starting state, which is showed in 1.

The reset method ensures proper environment initialization by setting up the board, tracking variables, and occasionally allowing the opponent to move first (though currently disabled with probability set to 0). It returns a TensorDict containing the initial observation, action mask indicating valid moves, step counter, and done flag, conforming to TorchRL's expected format for environment states.

1.1.2 Step Method Implementation

The `_step()` method forms the core of our environment, handling the game progression through these stages showed in 2:

This implementation ensures proper game flow while maintaining compatibility with TorchRL's training pipeline. The step function manages both player and opponent moves, implements a randomized success rate for piece placement, and calculates appropriate rewards based on game outcomes and board patterns.

1.1.3 Other Environment Components

Our implementation relies on several key components that work together with TorchRL to enable efficient agent training:

- **Observation and Action Space Specifications**: We define the environment's interface using TorchRL's specification system:
  - `observation_spec`: A `Composite` structure containing

Algorithm 1 Reset Method Implementation

```
function _RESET(tensordict, **kwargs)
    Initialize empty 12 × 12 board with zeros
    Set current_player ← 1                                   ▷ Player 1 starts
    winner ← None
    Clear move history and execution history
    step_count ← 0
    Load configuration parameters
    if random() < opponent_first_probability then            ▷ Rarely let opponent move first
        policy_net.eval()                                    ▷ Set policy network to evaluation mode
        With no gradient tracking:
            Get observation tensor
            Predict action using policy network
            Convert action index to board position
            Place opponent's piece on board
        policy_net.train()                                   ▷ Restore policy network to training mode
    end if
    observation ← _get_obs()                                 ▷ Get observation tensor from board state
    action_mask ← _get_action_mask()                         ▷ Generate valid action mask
    return TensorDict with observation, action_mask, step_count, done=False
end function
```

the board state (2×12×12 tensor), action mask (80-dimensional boolean tensor), and step counter

- `action_spec`: A `Categorical` specification with 80 possible actions
- `reward_spec`: A `Bounded` specification with range [-1.0, 1.0]

- **Action Mask Generation**: The `_get_action_mask()` method creates boolean masks that restrict the agent to valid moves, working with TorchRL's `MaskedCategorical` distribution to enable legal action selection.

- **Board Representation**: Our environment uses a specialized mapping system between action indices (0-79) and board positions through `index_to_position()` and `position_to_index()`, enabling efficient translation between TorchRL's action space and the internal board representation.

- **Winning Pattern Detection**: The winning conditions are analyzed through `_get_winning_combinations()` and `_check_winner()`, which determine terminal states and rewards that drive the agent's learning process.

These components ensure our environment is fully compatible with TorchRL's data collection and training pipeline while implementing the unique mechanics of our cross-shaped TicTacToe variant.

1.2 Policy and Value Network Design

Our agent's learning capability relies on two specialized neural networks: a policy network that determines action selections and a value network that estimates state values. Both networks share a similar architecture but serve distinct purposes in the reinforcement learning process.

1.2.1 Architecture Overview

Both networks follow a modern deep learning architecture with these key components:

- **Convolutional backbone:** Processes the 2-channel board representation
- **Residual blocks:** Facilitate information flow and gradient propagation
- **Specialized heads:** Task-specific final layers for policy or value prediction

1.2.2 Residual Block Implementation

We implemented residual blocks to improve training stability and convergence:

Algorithm 2 Step Method Implementation

```
function _STEP(tensordict)
    action ← tensordict["action"].item()
    i, j ← index_to_position(action)                    ▷ Convert action to board coordinates
    if random() < success_rate then                     ▷ Randomized move execution
        board[i][j] ← current_player
    else
        Select random adjacent position if possible
    end if
    winner ← _check_winner()                            ▷ Check if game is over
    if winner == current_player then
        reward ← 1.0
        done ← true
    else if is_over() then                              ▷ Board is full (draw)
        reward ← 0.0
        done ← true
    else                                                ▷ Opponent's turn
        opponent_obs ← -_get_obs()                      ▷ Negate observation for opponent
        opponent_logits ← policy_net(opponent_obs["logits"]
        Apply action mask to opponent_logits
        action ← sample from Categorical distribution
        board[i][j] ← -current_player                   ▷ Opponent makes move
        winner ← _check_winner()                        ▷ Check if opponent won
        if winner != null then
            if winner == -current_player then
                reward ← -1.0
            else                                        ▷ Draw
                reward ← 0.0
            end if
            done ← true
        else                                            ▷ Game continues
            reward ← calculate_pattern_reward(current_player) * 0.1
            done ← false
        end if
    end if
    steps ← steps + 1
    return TensorDict with observation, reward, done, action_mask, step_count
end function
```

$$\text{ResBlock}(x) = \text{ReLU}(x + \text{GN}(\text{Conv}(\text{ReLU}(\text{GN}(\text{Conv}(x))))))) \quad (1)$$

Each residual block preserves input dimensions and includes:

- Two 3×3 convolutional layers with same-padding
- Group normalization layers for training stability
- Skip connections to improve gradient flow
- ReLU activations between layers

1.2.3 Policy Network

The policy network produces action probabilities over the 80 valid board positions:

$$Policy(logits_a|s) = \text{Softmax}\Big(\text{FC}\big(\text{Flatten}(\text{Conv}_{\text{policy}}(\text{ResBlocks}(\text{Conv}_{\text{base}}(s))))\big)\Big) \quad (2)$$

Notable features include:

- Input: 2×12×12 tensor representing the board state
- Base convolution with 128 filters and normalization
- Four residual blocks for feature extraction
- Policy-specific convolution with 32 filters
- Fully connected output layer with 80 units (one per valid action)
- **Position-based action mask weighting** to encourage strategically valuable moves, which will be elaborated later.

1.2.4 Value Network

The value network estimates the expected outcome from the current state:

$$V(s) = \sigma\Big(\text{FC}\big(\text{Flatten}(\text{Conv}_{\text{value}}(\text{ResBlocks}(\text{Conv}_{\text{base}}(s))))\big)\Big) \quad (3)$$

Key components include:

- Identical backbone architecture to the policy network
- Value-specific convolutional layer with 32 filters
- Single-output fully connected layer to produce scalar value
- Sigmoid activation to bound output between 0 and 1

1.2.5 Position-Based Action Weighting

As we mentioned in 1.2.3, we incorporated domain knowledge into the policy network through a position-based weighting mechanism:

$$Policy_{final}(logits_a|s) = \text{Normalize}\Big(Policy(logits_a|s) \cdot \big(w(a,s) + 0.1\big)\Big) \quad (4)$$

Where $w(a, s)$ calculates potential scores for each position based on:

- Adjacency to existing pieces
- Potential to form winning patterns
- Normalized to preserve action distribution shape

This key innovation in our policy network architecture is the incorporation of a strategic action weighting mechanism using the `weight_mask` calculated from the board state. This mask introduces domain-specific knowledge by assigning different importance weights to different board positions based on their strategic value. After generating the initial logits through the neural network, we apply a post-processing step: (1) normalizing the raw logits using softmax, (2) normalizing the weight mask to a [0,1] range to ensure compatibility, and (3) combining them through multiplication with a small constant added to preserve some of the original policy distribution. This approach effectively guides the agent toward promising moves while still allowing it to explore the policy space learned from training data. The final combined distribution is converted back to log space before being returned as the action logits. This mechanism significantly accelerates learning by focusing the agent's attention on strategically valuable positions, particularly early in training when random exploration would be inefficient on our large cross-shaped board.

1.3 Reward Design

Our reward function R combines terminal and intermediate rewards to provide effective learning signals:

$$R = \begin{cases} 1.0, & \text{if agent wins} \\ -1.0, & \text{if agent loses} \\ 0.0, & \text{if draw} \\ 0.1 \cdot R_{pattern}, & \text{otherwise} \end{cases} \tag{5}$$

The pattern reward $R_{pattern}$ is calculated as:

$$R_{pattern} = \sum_{l \in \{2,3,4\}} w_l \cdot count_l \tag{6}$$

where $count_l$ is the number of detected patterns of length l, and w_l is the corresponding weight:

$$w_2 = 0.2 \text{ for 2-in-a-row} \tag{7}$$

$$w_3 = 0.5 \text{ for 3-in-a-row} \tag{8}$$

$$w_4 = 0.8 \text{ for 4-in-a-row} \tag{9}$$

This reward structure encourages both strategic play through terminal rewards and tactical positioning through intermediate pattern recognition.

1.4 PPO Pipeline with TorchRL

Our implementation leverages TorchRL's modular components to create an efficient and flexible PPO training pipeline. Each component serves a specific purpose in the reinforcement learning process, working together to enable effective policy optimization.

1.4.1 Actor-Critic Architecture

The actor-critic framework is implemented using TorchRL's modular design:

- **Policy Module**: We wrap our policy network with `TensorDictModule`, defining the interface for processing observations into action logits. This module transforms the 2D board representation into a probability distribution over the 80 possible actions.
- **Probabilistic Actor**: The `ProbabilisticTensorDictModule` transforms logits into a proper probability distribution, applying the action mask to ensure only valid moves are considered. We use the specialized `MaskedCategorical` distribution class that enforces zero probability for invalid actions while properly normalizing the distribution over valid actions.
- **Value Network**: Implemented as a `ValueOperator`, this module estimates state values needed for advantage calculation, sharing the same observation interface as the policy network but producing scalar value estimates instead of action probabilities.

1.4.2 Data Collection and Processing

The data pipeline efficiently gathers experiences and prepares them for training:

- **Synchronous Collector**: The `SyncDataCollector` manages environment interaction, using our environment factory to create instances and collect trajectories according to the current policy. It handles batching of experiences and maintains proper episode boundaries for more stable learning.
- **Replay Buffer**: Though PPO is typically on-policy, we use a `ReplayBuffer` with `LazyTensorStorage` to efficiently manage the collected experiences during each update cycle. We clear this buffer between collection phases to maintain the on-policy nature of PPO while enabling multiple passes over the same data during optimization.

1.4.3 Advantage Estimation and Loss Calculation

The core of PPO's learning mechanism is implemented through:

- **Generalized Advantage Estimation**: The `GAE` module calculates advantages with configurable discount factor (γ) and GAE parameter (λ), reducing variance in policy gradient estimates while maintaining an acceptable bias level. This balance is crucial for stable learning in our stochastic game environment.
- **PPO Loss**: The `ClipPPOLoss` implements PPO's clipped objective function, combining policy loss, value function loss, and optional entropy bonus. The clipping mechanism prevents excessively large policy updates, maintaining proximity to the data distribution and improving training stability.

1.4.4 PPO Loss Formulation

The PPO loss implemented by TorchRL is as below:

$$L^{\text{total}} = L^{\text{policy}} + c_1 \cdot L^{\text{value}} - c_2 \cdot L^{\text{entropy}} \quad (10)$$

The policy loss uses the clipped surrogate objective:

$$L^{\text{policy}}(\theta) = -\mathbb{E}_t \Big[\min \Big(r_t(\theta) \cdot \hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) \cdot \hat{A}_t \Big) \Big] \quad (11)$$

where $r_t(\theta)$ is the probability ratio:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \quad (12)$$

The value function loss uses smooth L1 loss:

$$L^{\text{value}}(\theta) = \mathbb{E}_t \left[\text{SmoothL1} \left(V_\theta(s_t) - V_t^{\text{target}} \right) \right] \quad (13)$$

The entropy bonus encourages exploration:

$$L^{\text{entropy}}(\theta) = \mathbb{E}_t \left[H(\pi_\theta(\cdot|s_t)) \right] \quad (14)$$

The coefficients c_1 and c_2 control the relative importance of the value loss and entropy bonus, respectively, while ϵ determines the clipping range.

This integrated pipeline enables efficient training while maintaining the flexibility to adjust hyperparameters and module configurations based on experimental results.

1.5 Training Process

The training loop (showed as 3) follows a three-tiered structure that implements the PPO algorithm with experience collection and multi-epoch updates. First, the collector gathers experiences by interacting with the environment. Then, advantage estimates are computed to determine how much better each action performed compared to expected values. Finally, the policy and value networks are updated through multiple passes over minibatches of the collected data.

1.6 Experiment Configuration

The PPO algorithm is configured with the following parameters:

- **Learning Rate**: 3×10^{-4} with 10^{-4} weight decay for stable gradient updates

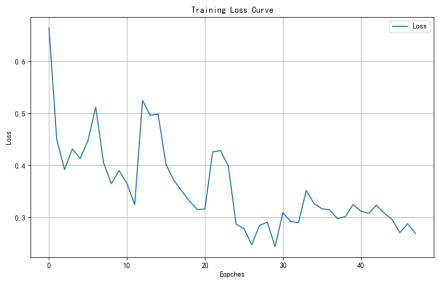

Figure 1: training loss

- **Discount Factor**: $\gamma = 0.99$ balances immediate and future rewards
- **GAE Parameter**: $\lambda = 0.95$ reduces variance in advantage estimation while maintaining acceptable bias
- **Clipping Threshold**: $\epsilon = 0.2$ prevents destructively large policy updates
- **Loss Coefficients**: Value loss weighted by $c_1 = 0.2$ and entropy bonus by $c_2 = 0.02$

2 Experiments and Results

2.1 Experimental Setup

All experiments employed the Proximal Policy Optimization (PPO) algorithm with consistent hyperparameters: 4 residual blocks (128 filters), batch size of 128, and learning rate 0.0003. Training utilized a 12×12 cross-shaped board with 80 valid actions.

2.2 Convergence Analysis

Figure 1 shows the training loss curve over the course of training. The total loss starts high (approximately 0.67) and generally decreases over time, converging to around 0.28 after 50 epochs. The initial high volatility gradually stabilizes, indicating the agent is finding a consistent policy.

Figure 2 breaks down the total loss into its components: policy loss, value loss, and entropy loss. We observe that:

- The policy loss (orange) initially drops rapidly before stabilizing, indicating quick initial learning of basic tactics
- The value loss (green) fluctuates more, reflecting the challenge in accurately evaluating complex board positions
- All components show general convergence in later epochs

Algorithm 3 PPO Training Loop

for batch $i = 1$ to $N_{bigbatches}$ **do** ▷ each big batch contains N_{frames} frames(i.e. steps)
    Collect experiences using current policy
    Compute advantage estimates for all experiences
    for epoch $j = 1$ to N_{epochs} **do** ▷ go through the big batch for $N_{epoches}$ times
        for minibatch $k = 1$ to $N_{frames}/N_{minibatchsize}$ **do**
            Sample minibatch of transitions
            Compute policy, value, and entropy losses
            Update networks using gradient descent
        end for
    end for
    Update learning rate
end for

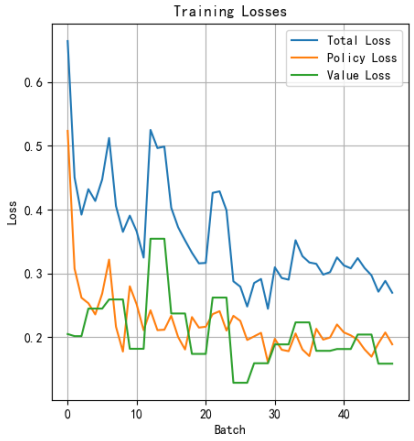

Figure 2: Component Losses

The model was trained for a total of 50 epochs, with checkpoints saved at regular intervals and whenever a new best performance was achieved. Our best-performing model achieved a 100% success rate against random opponents, demonstrating the effectiveness of our training approach.

2.3 Experimental Results

Our experiments show a clear progression in agent performance across different training configurations, demonstrating the impact of key design choices on learning effectiveness:

The results showed in 1 highlight several important findings:

- **Opponent Quality**: Comparing Training 3 and Training 4, which differ only in opponent strategy, we observe that self-play using the policy network as an opponent increases the success rate from 62% to 67%. This demonstrates that training against a more strategic opponent (even if initially weak) provides better learning signals than facing random moves, as it encourages the agent to develop more robust strategies rather than exploiting random behavior.

- **Reward Engineering**: The influence of reward structure is evident when comparing Training 4 and Training 5, where changing from the original sparse rewards (+1/-1/0/0.1) to a designed reward system incorporating pattern recognition for consecutive stone formations in- creases performance from 67icant improvement demonstrates that providing intermediate rewards for strategically valuable board configurations helps guide exploration and acceler- ates learning of effective policies.

- **State Representation**: Expanding from a single-channel to a two-channel representation (separately encoding player and opponent pieces) significantly improved learning efficiency, reaching 85% success rate with just 10 training epochs.

- **Action Weighting**: Our position-based action weighting mechanism greatly accelerated learning by focusing exploration on strategically valuable moves, particularly important for our large cross-shaped board.

- **Training Duration**: The final rows of Table 1 demonstrate the importance of sufficient training time. While our enhanced configuration in Training 10 already performs well with 10 epochs (85%), extending training to 20 epochs dramatically increases performance to 99%, with perfect performance (100%) achieved at 50 epochs. This suggests that while our ar-

Table 1: Success rates for different training configurations

Configuration	State Representation	Reward Structure	Opponent Type	Success Rate (%)
Training 3	1-channel	Original	Random	62
Training 4	1-channel	Original	Policy	67
Training 5	1-channel	Designed	Policy	76
Training 10 (10 epochs)	2-channel + Weighting	Designed	Policy	85
Training 10 (20 epochs)	2-channel + Weighting	Designed	Policy	99
Training 10 (50 epochs)	2-channel + Weighting	Designed	Policy	100

chitectural improvements enhance learning efficiency, allowing sufficient time for policy refinement remains crucial.

These results confirm that our final approach effectively combines improved state representation, strategic reward design, self-play opponent modeling, and knowledge-guided exploration achieves an better performance in this complex variant of Tic-Tac-Toe.

3 AlphaZero for Standard TicTacToe

We also implemented AlphaZero for the standard 3*3 tictactoe game without any 3rd-party RL frameworks or libraries. In this section, we will introduce our implementation in details.

3.1 Environment: TicTacToe Class

Unlike our PPO implementation that relies on TorchRL, our AlphaZero environment is self-implemented with several distinctive features:

- **State Representation**: The game state is represented as a simple 3×3 NumPy array with values {-1, 0, 1} indicating opponent pieces, empty spaces, and player pieces respectively.
- **Action Space**: Actions are represented as coordinate pairs (i, j) rather than flattened indices, simplifying the mapping between board positions and moves.
- **Stochastic Transitions**: Similar to our PPO environment, the environment implements a stochastic transition model where each action has a configurable probability (`success_prob`) of succeeding or otherwise landing in an adjacent position.
- **State Cloning**: The environment supports creating independent deep copies through the `dcopy()` method, which is essential for MCTS to run simulations without modifying the actual game state.

This streamlined design enables efficient tree search operations while maintaining the key MDP properties needed for reinforcement learning.

3.2 Policy and Value: AlphaZeroNet Class

The neural network is a critical component of AlphaZero, serving as both a policy evaluator and a value estimator. This dual-headed design is central to AlphaZero's architecture and performance.

3.2.1 Network Architecture

AlphaZero employs a deep neural network with a shared feature extraction backbone and two separate output heads for policy and value prediction:

- **Shared Representation:** A series of convolutional layers process the board state to extract meaningful spatial features
- **Policy Head:** Projects the shared features to a probability distribution over all possible actions
- **Value Head:** Estimates the expected outcome of the current board position

The network architecture for our TicTacToe implementation consists of:

- An initial convolutional layer that processes the game state representation
- Multiple residual blocks that provide deep feature extraction while avoiding gradient degradation
- A policy head that outputs action probabilities
- A value head that provides state evaluation in the range $[-1, 1]$

3.2.2 State Representation

The network input uses a 3-channel representation:

- Channel 1: Current player's pieces (binary)
- Channel 2: Opponent's pieces (binary)
- Channel 3: Player indicator (constant plane of 1 or -1)

This representation encodes both the board position and the player to move, allowing the network to understand the game context fully.

3.2.3 Policy Head

The policy head transforms shared features into action probabilities:

- 1×1 convolutional layer reduces the channel dimension while preserving spatial information
- Batch normalization layer stabilizes training
- Fully connected layer maps to action space dimensionality
- Softmax activation ensures a valid probability distribution

During MCTS, these probabilities serve as priors that guide the search, focusing exploration on promising actions.

3.2.4 Value Head

The value head estimates the expected outcome from the current position:

- 1×1 convolutional layer reduces feature dimensions
- Two fully connected layers with ReLU activation
- Tanh activation in the final layer bounds output to $[-1, 1]$

The value $[-1, 1]$ represents the expected outcome from the current player's perspective, with +1 indicating a certain win, -1 a certain loss, and 0 suggesting a draw.

3.2.5 Training Objective

The network is trained to minimize a combined loss function:

$$\mathcal{L} = \mathcal{L}_\pi + \mathcal{L}_v \tag{15}$$

$$\mathcal{L}_\pi = -\frac{1}{n}\sum_{i=1}^{n}\sum_{a\in\mathcal{A}} \pi_i(a)\log(p_i(a)+\epsilon) \tag{16}$$

$$\mathcal{L}_v = \frac{1}{n}\sum_{i=1}^{n}(z_i - v_i)^2 \tag{17}$$

where: π_i is the target policy distribution from MCTS visit counts, p_i is the policy output from neural network, z_i is the game outcome from player's perspective ($+1$ win, 0 draw, -1 loss), v_i is the value prediction from neural network, ϵ is a small constant (10^{-10}) to prevent $\log(0)$, n is the batch size.

This joint training approach allows the network to simultaneously improve at predicting both action probabilities and position evaluation, with each task benefiting from the shared feature extraction.

3.3 MCTS: Monte Carlo Tree Search Implementation

Monte Carlo Tree Search (MCTS) is a fundamental component of AlphaZero. Our implementation consists of two main classes: `MCTSNode` and `MCTS`, which work together to explore the game tree efficiently.

3.3.1 Overview of MCTS Algorithm

MCTS builds a partial game tree guided by both simulation results and neural network predictions. Unlike traditional MCTS which relies on random rollouts, our AlphaZero-style implementation uses the neural network to evaluate positions and guide search.

The algorithm iteratively performs four stages:

1. **Selection:** Starting from the root node, traverse the existing tree by selecting child nodes based on a UCB-like formula until reaching a leaf node.
2. **Expansion:** Add child nodes of the root for each possible action.
3. **Evaluation:** Instead of random rollouts, use the neural network to evaluate the position.
4. **Backpropagation:** Update statistics for children nodes which were visited.

3.3.2 MCTSNode Class

The `MCTSNode` class represents a node in the Monte Carlo search tree, storing essential statistics and managing the tree structure.

Key Attributes Each node contains several important attributes:

- **Children:** A dictionary mapping valid moves to corresponding child nodes, with structure `{(i,j): MCTSNode, ...}`
- **Q:** The average action value, representing expected outcome from this node
- **N:** Visit count, tracking how many times this node has been traversed
- **P:** Prior probability from the policy network, used to guide exploration
- **W:** Win count, tracking how many simulations resulted in a win
- **Vs:** A list recording evaluation values whenever this node is selected

It's worth noting that for the root node (representing the current state), these statistics are primarily meaningful for its children (representing possible actions).

Node Selection Child selection uses an Upper Confidence Bound (UCB) formula adapted for AlphaZero:

$$\text{UCB}(s,a) = Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \tag{18}$$

Where:

- $Q(s,a)$ is the estimated value of action a in state s
- $P(s,a)$ is the prior probability from the policy network
- $N(s,a)$ is the visit count
- c_{puct} is a constant that controls exploration

This formula balances exploitation (first term) and exploration (second term), with the exploration term weighted by the policy network's prior probability.

Node Expansion When expanding a node (a state), we create a root node for the state and child nodes for all legal actions from this state, with each action node initialized with the prior probability from the policy network. These priors may be adjusted with Dirichlet noise at the root to encourage exploration during self-play.

Value Updating After each simulation, we update the statistics of the visited node:

- Increment the visit count N
- If simulation resulted in a win, increment the win count W
- Record the value prediction combined with the simulation result as $V_s = 0.5 \times (v + r)$, where v is the value network prediction and r is the simulation result
- Update Q as the mean of all recorded values in V_s

Policy Generation The final policy distribution is derived from visit counts using a temperature parameter:

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \tag{19}$$

Where τ is the temperature parameter that controls the exploration versus exploitation tradeoff. Lower temperature leads to more focused selection of the best actions.

3.3.3 MCTS Class

The `MCTS` class orchestrates the search process, managing simulations and interactions with the neural network.

Search Algorithm Below is the pseudocode for our MCTS implementation: 4

Key Implementation Details Several important features distinguish our MCTS implementation:

1. **Neural Network Guidance:** Instead of random rollouts, we use the neural network to evaluate positions and guide simulations.
2. **Value Fusion:** We combine the neural network's value prediction with the simulation result, allowing the search to benefit from both immediate evaluation and complete playouts.

Algorithm 4 AlphaZero-style MCTS

```
function RUN(game)
    root ← MCTSNode()
    legal_moves ← game.get_legal_moves()
    probs, _ ← neural_network(game_state)
    Apply mask to keep only legal moves in probs
    If enabled, apply Dirichlet noise to probs
    root.expand(legal_moves, probs)
    for i = 1 to num_simulations do
        sim_result, choice ← Simulate(game.copy(), root)
        root.update_child(sim_result, choice)
    end for
    return root
end function
function SIMULATE(game, node)
    simulating_player ← game.current_player
    move ← node.select_child(c_puct)
    game.execute_move(move)
    _, value ← neural_network(game_state)
    while not game.is_over() do
        probs, _ ← neural_network(game_state)
        Apply mask to keep only legal moves
        move ← argmax(probs)
        game.execute_move(move)
    end while
    if simulating_player = game.winner then
        sim_result ← 1
    else if game.winner = 0 then
        sim_result ← 0
    else
        sim_result ← −1
    end if
    node.children[move].Vs.append(0.5 × (value + sim_result))
    return sim_result, move
end function
```

3. **Modified Backpropagation:** Rather than updating all nodes in the path, we directly update the statistics of the child node chosen from the root, since we're primarily interested in selecting the best immediate action.

4. **Simulation Strategy:** After selecting the initial move from the root, subsequent moves in the simulation follow a greedy policy using the neural network's predictions rather than continuing tree search.

3.4 Trainer: Self-Play and Neural Network Training

The Trainer class orchestrates the entire AlphaZero learning process, managing the interaction between self-play data generation and neural network training.

3.4.1 Trainer Design Overview

Our Trainer implementation follows the fundamental AlphaZero training methodology with several key components:

- **Data Generation:** Executing self-play games using the current neural network and MCTS to create high-quality training examples.
- **Memory Buffer:** Maintaining a fixed-size replay buffer that stores state-action-outcome triplets from self-play games.
- **Network Training:** Sampling batches from the replay buffer to improve the neural network's policy and value predictions.
- **Iterative Improvement:** Repeating this process to progressively enhance the neural network's gameplay ability.

3.4.2 Self-Play Data Generation

The self-play procedure is a critical component that generates training data by having the neural network play against itself:

Game Execution Process For each position encountered during self-play:

1. Run MCTS using the current neural network to obtain improved policy probabilities.
2. Record the current state, MCTS-derived policy distribution, and current player.
3. Select a move based on the MCTS visit counts, moderated by a temperature parameter.
4. Execute the selected move and continue until the game concludes.
5. When the game ends, record the outcome as a value target for each saved state.

Training Example Generation For each completed game, training examples are created with the following structure:

- **Input:** Board state representation (s_t)
- **Policy Target:** Improved policy distribution from MCTS (π_t)
- **Value Target:** Game outcome from current player's perspective (z_t)

The value target reflects whether the current player at state s_t ultimately won the game (+1), lost the game (-1), or reached a draw (0).

3.4.3 Neural Network Training

The training process uses supervised learning to update the neural network based on the collected self-play data:

Batch Sampling Rather than training on sequential moves, the Trainer randomly samples a mini-batch of experiences from the replay buffer. This approach:

- Reduces correlation between training samples
- Improves the stability of the learning process
- Prevents the network from overfitting to specific game trajectories

3.4.4 Training Loop Algorithm

The complete AlphaZero training process is summarized in the following pseudocode: 5

This creates a virtuous cycle where the agent continuously improves without any external knowledge or human guidance, bootstrapping its way to mastery through pure self-play.

3.5 Experiment Results

Due to the limitation of time, we only implemented a training code for AlphaZero version and did not implement a test code. If we have time after exams we may update our test results to github.

Algorithm 5 AlphaZero Training Loop

Initialize neural network with random weights θ
Initialize replay buffer $\mathcal{D}$ with capacity N
for iteration $= 1$ to $n\_iterations$ **do**
    Reset replay buffer for current iteration
    for game $= 1$ to n_games **do**
        game_examples $\leftarrow$ Empty list
        Initialize game state s
        while game not over **do**
            root $\leftarrow$ MCTS.run(game, neural_network, $n\_simulations$)
            $\pi \leftarrow$ root.get_pi() ▷ Extract policy from visit counts
            record(game_state, π, current_player)
            $a \leftarrow$ SampleAction(π, temperature)
            Execute action a in game
        end while
        winner $\leftarrow$ game.winner
        for each (state, π, player) in game_examples **do**
            if player $=$ winner **then**
                $z \leftarrow 1$
            else if winner $= 0$ **then**
                $z \leftarrow 0$
            else
                $z \leftarrow -1$
            end if
            Add (state, π, z) to replay buffer $\mathcal{D}$
        end for
    end for
    if $|\mathcal{D}| \geq$ batch_size **then**
        for epoch $= 1$ to n_epochs **do**
            Sample minibatch B from replay buffer $\mathcal{D}$
            Update network parameters θ using gradient descent:
            $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(B)$
        end for
    end if
    if iteration mod save_frequency $= 0$ **then**
        Save network parameters θ
    end if
end for

4 Conclusion

We implemented this game through 2 different algorithms: PPO and AlphaZero.

For PPO approach, we investigated the implementation of PPO pipiline of TorchRL framework by reading their source code in detail. We could see that after training our agent is much more efficient to win the game. We also compared several configurations we tryed during our experiments, proving the effectiveness of the final version we adopted.

For AlphaZero, we learned the details of the algorithm and implemented it by ourselves. The debugging was struggling but meaningful. Hope we could explore more on RL in our future studying or working.