Hanoi University of Science and Technology

School of Information and
Communication Technology

Project Report:
TicTacToe Solvers

Authors:
Tô Duy An

Student ID:
202416653

January 10, 2026

Contents

Abstract

This report explores various strategies for training agents to play TicTacToe and Gomoku, utilizing approaches such as heuristic-based methods, tree search algorithms, and reinforcement learning techniques. It also details the creation of custom datasets, model architectures, and experimental setups. The results are mixed: some agents, like Monte Carlo Tree Search (MCTS) and Alpha-Beta pruning, demonstrate strong performance, while others, such as neural network-based MCTS, show promising but sub-optimal outcomes.

All code and implementation please check out this link

Chapter 1

Introduction

Board games have long served as the primary testbed for Artificial Intelligence (AI), providing structured environments to evaluate decision-making capabilities. Among these, TicTacToe stands as a foundational cornerstone. While its 3x3 grid and limited state space make it computationally trivial for modern systems, it remains an ideal environment for visualizing and understanding core game theory concepts.

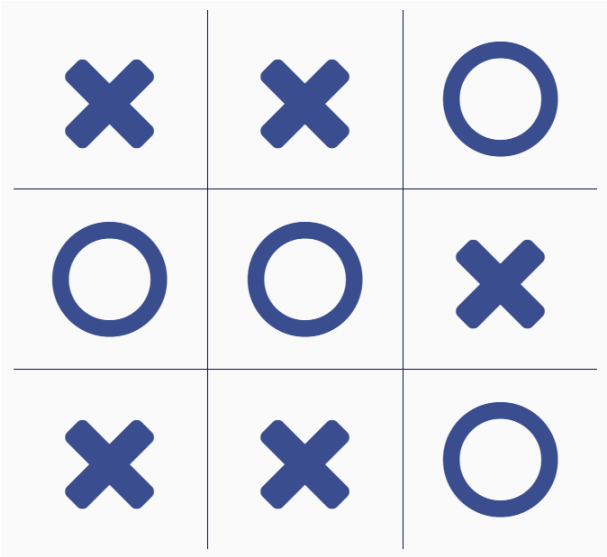

Figure 1.1: A sample TicTacToe complete game state

The history of algorithms for playing TicTacToe dates back to early implementations of minimax, a decision-making algorithm that evaluates possible moves to maximize a player's advantage while minimizing the opponent's. Over time, advancements such as Alpha-Beta pruning, Monte Carlo Tree Search (MCTS), and heuristic-based methods have been developed to improve efficiency and scalability. More recently, reinforcement learning and neural network-based approaches, inspired by AlphaGo, have been applied to larger board variants like Gomoku, which extend the challenge to a 10x10 grid and require aligning five marks to win.

This project delves into these strategies, exploring their strengths and limitations while training agents to play TicTacToe and Gomoku. It begins with creating custom datasets, replicating game environments, and building models to overcome the challenges posed by these games.

Chapter 2

Game engine and encoding game state

In this project, I only consider square board for tictactoe and Gomoku. While other shape of the board could also be play on, by making a board square, we can make use of symmetry to reduce possible board state to be stored.

2.1 Symmetry

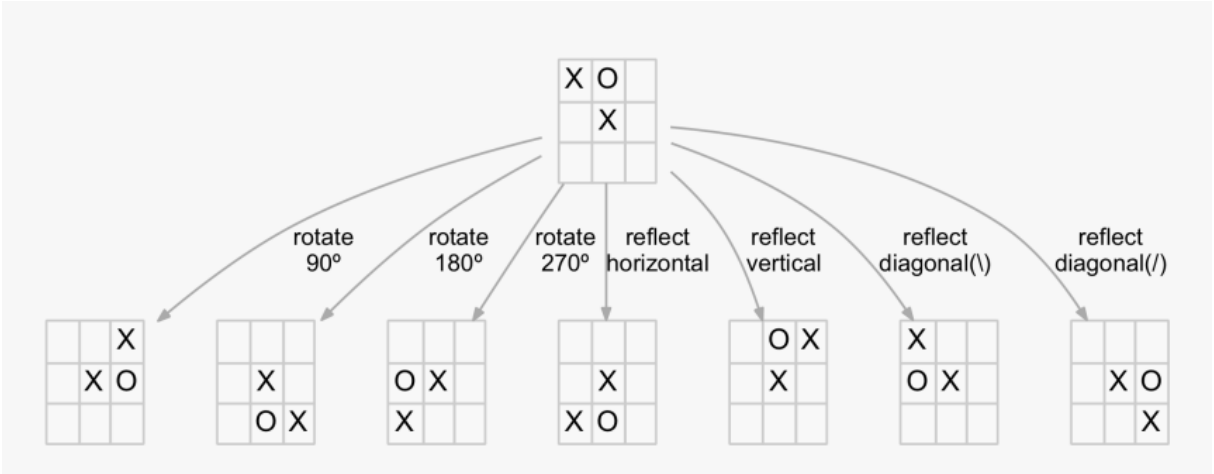

Figure 2.1: Possible duplicate game states

In TicTacToe and Gomoku, the symmetry of the square board introduces the possibility of duplicate game states. These duplicates arise when the board is rotated, as the relative positions of marks remain unchanged in terms of gameplay logic. For example, a board rotated 90°, 180°, or 270° clockwise represents the same strategic state as the original configuration. Some key observation:

- Original orientation: The board remains unchanged.
- 90° clockwise rotation: The rows become columns, and the order is reversed.
- 180° rotation: The board is flipped upside down.
- 270° clockwise rotation: Similar to 90°, but reversed further.
- Mirror: The board is mirrored along the middle row

By combining the rotation and mirror, we can achieve 8 game duplicate game states that can be derive from the original.

2.2 Encoding

To encode board states in our code, we represent the board as a 2D array or matrix, where each cell holds a value indicating its state:

- 0: Empty cell
- 1: Player X's mark
- -1: Player O's mark

In games played on a square grid like Tic-Tac-Toe or Gomoku, the state space contains many isomorphic states—positions that are strategically identical but differ only by rotation or reflection. To handle this, we implemented a canonical encoding system. For any given board configuration, we generate all 8 possible symmetry transformations:

- 4 Rotations ($0°, 90°, 180°, 270°$)
- 4 Reflections (Mirroring combined with the rotations)

We convert these transformed states into string representations and select the lexicographically smallest one as the **Canonical Form**. By storing and training on only the canonical form, we significantly reduce the size of the state space and improve the learning rate of our agent, as knowledge learned from one position is immediately applicable to its 7 symmetrical counterparts. When we query a board state later, we also return a rotation key to help the agent select the correct move.

Algorithm 1 Board State Encoding Strategies

Input: C: 1D array of board cells (Current State)
Input: S: Symbol of current player (X or O)
Input: $Rows, Cols$: Dimensions of the board
Reduces state space by treating all rotations/reflections as identical.

procedure GetCanonicalHash(C)
  $M \leftarrow \text{Reshape}(C, (Rows, Cols))$
  $Symmetries \leftarrow \emptyset$
  for $k \in \{0, 1, 2, 3\}$ **do** ▷ Generate all 8 symmetries
    $T_{rot} \leftarrow \text{Rotate90}(M, k)$
    $Symmetries.\text{append}(\text{Flatten}(T_{rot}))$
    $T_{flip} \leftarrow \text{Rotate90}(\text{Flip}(M), k)$
    $Symmetries.\text{append}(\text{Flatten}(T_{flip}))$
  end for
  $C_{canonical} \leftarrow \min_{lex}(Symmetries)$ ▷ Select lexicographically smallest form and hash
  $H_{canonical} \leftarrow \text{Hash}(\text{Tuple}(C_{canonical}))$
  return $H_{canonical}$
end procedure

Chapter 3

Conventional Solution

This chapter examines three foundational adversarial search strategies: Minimax, Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS). While Minimax establishes a theoretical baseline through exhaustive game tree traversal to guarantee optimal play, its exponential time complexity renders it impractical for larger state spaces. Alpha-Beta Pruning addresses this inefficiency by eliminating redundant branches to accelerate inference without compromising accuracy, whereas MCTS introduces a probabilistic, simulation-based paradigm that scales effectively to complex domains like Gomoku where deterministic search is computationally prohibitive.

3.1 Mini Max and Alpha beta pruning

The logic of Minimax and Alpha beta pruning represents the "baseline" solution. The central idea is that the AI does not simply guess; it simulates the future, searching for a solution.

3.1.1 Mini Max logic

The code treats the game as a recursive tree. The function `minimax` takes the current board state and "plays out" every remaining game until it hits a terminal condition (Win, Loss, or Draw).

- **The Simulation Engine:** The `Board` class is immutable during simulation. When the code calls `board.act()`, it generates a *new* board instance. This allows the recursive function to explore millions of hypothetical futures without corrupting the actual game being played.
- **The Decision Rule:** The algorithm assumes the opponent is rational.
  - When it is the AI's turn (Maximizing), it loops through all valid moves and picks the one with the highest score ($+1$).
  - When it is the Opponent's turn (Minimizing), the AI assumes they will pick the move that gives the AI the lowest score (-1).

While this guarantees a perfect defense, the implementation in `minimax.py` is "greedy" for information—it checks *every* single possibility, even the obviously bad ones.

3.1.2 Alpha-Beta logic

The transition to `alpha_beta.py` was driven by a simple observation: **If we found a winning move, why keep searching.**

Imagine the AI is considering three moves: A, B, and C.

1. The AI analyzes Move A and discovers it leads to a forced Win.
2. In standard Minimax, the AI would still pause to painstakingly analyze Move B and Move C.
3. This is wasteful. Since a Win is the best possible outcome, moves B and C are irrelevant.

This logic gave birth to the solution in `alpha_beta.py`. We introduced two variables to track "guarantees":

- α (Alpha): The best score the AI can guarantee so far.
- β (Beta): The worst score the Opponent can force the AI into.

Using these two value, the optimization happens inside the loop when:

```
if beta <= alpha:
    break
```

This single conditional statement is the difference between the two files. It tells the algorithm a line of play that proves this specific branch is worse than a move we examined earlier and stop search.

3.1.3 Complexity Analysis

Minimax

The current implementation performs a complete Depth-First Search.

- **Time Complexity:** $O(b^d)$
- **Explanation:** With a branching factor b (legal moves) and depth d (turns remaining), the algorithm visits every leaf node. In Tic-Tac-Toe, roughly 255,168 leaf nodes are possible in the worst case. While feasible here, this complexity makes the algorithm useless for games like Chess or Go.

Alpha-Beta

The `alpha_beta.py` implementation prunes the tree.

- **Time Complexity:** $O(b^{d/2})$ (in the best case).
- **Explanation:** By rejecting bad branches early, the effective branching factor is reduced to $\sqrt{b}$. This means the Alpha-Beta agent can search twice as deep as the Minimax agent in the same amount of time.

Conclusion

The code evolved from a mathematically correct but inefficient brute-force search (`minimax`) to a pragmatic, optimized search (`alpha_beta`). Both solutions are logically sound and result in an unbeatable Tic-Tac-Toe agent, but the Alpha-Beta implementation represents the scalable approach required for complex decision-making environments.

3.1.4 Pseudo code

Algorithm 2 Minimax with Alpha-Beta Pruning

Input: $Board$: Current board state
Input: α: Best value max player can guarantee
Input: β: Best value min player can guarantee
Input: $Player$: The player whose turn it is
Input: $RootPlayer$: The maximizing agent (AI)
Output: Best heuristic score for the current board state

```
1:  procedure AlphaBeta(Board, α, β, Player, RootPlayer)
2:      Result ← Board.isEnd()
3:      if Result ≠ Incomplete then
4:          return OutcomeScore(Result, RootPlayer)
5:      end if
6:      if Player == RootPlayer then                          ▷ Maximizing Step
7:          Best ← −∞
8:          for Move ∈ Board.get_valid_moves() do
9:              Child ← Board.act(Move, Player)
10:             NextPlayer ← Switch(Player)
11:             Val ← AlphaBeta(Child, α, β, NextPlayer, RootPlayer)
12:             Best ← max(Best, Val)
13:             α ← max(α, Val)
14:             if β ≤ α then                                 ▷ Beta Cut-off
15:                 break
16:             end if
17:         end for
18:         return Best
19:     else                                                  ▷ Minimizing Step (Opponent)
20:         Best ← +∞
21:         for Move ∈ Board.get_valid_moves() do
22:             Child ← Board.act(Move, Player)
23:             NextPlayer ← Switch(Player)
24:             Val ← AlphaBeta(Child, α, β, NextPlayer, RootPlayer)
25:             Best ← min(Best, Val)
26:             β ← min(β, Val)
27:             if β ≤ α then                                 ▷ Alpha Cut-off
28:                 break
29:             end if
30:         end for
31:         return Best
32:     end if
33: end procedure
```

3.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm for decision processes, most notably employed in game play. Unlike Minimax, which requires a heuristic evaluation function for non-terminal states, MCTS estimates the value of a state by running many random simulations (playouts) to the end of the game.

My implementation adapts the standard MCTS architecture for Tic-Tac-Toe. It introduces a critical optimization: **Canonical State Deduplication**. By recognizing that rotated or reflected board states are strategically identical, the algorithm significantly reduces the search space, allowing for faster convergence to optimal play.

3.2.1 Node Representation

The tree consists of `Node` objects that store statistical data accumulated during simulations. Each node tracks:

- **Visit Count (N):** How many times this state has been explored.
- **Outcomes:** The number of Wins (W), Draws (D), and Losses (L) observed from this state.
- **Exploration Constant (C):** A parameter tuning the balance between visiting new states and exploiting known good states (default 1.4).

3.2.2 Transposition Table (Deduplication)

A standard MCTS builds a tree where identical board states reached via different move orders create distinct nodes. This implementation uses a dictionary `self.nodes` to enforce a **Directed Acyclic Graph (DAG)** structure.

Before creating a new node, the method `get_createNode` calculates the *Canonical Form* of the board (the lexicographically smallest symmetry).

```
key = str(canonical_cells)
if key not in self.nodes:
    self.nodes[key] = Node(...)
return self.nodes[key]
```

This ensures that all symmetric variations of a state map to a single shared node, allowing the agent to learn from transpositions effectively.

3.2.3 The Four Phases of MCTS

The `search` method executes the four standard phases of MCTS iteratively.

Selection

Starting from the root, the algorithm traverses down the tree by selecting the child node that maximizes the **Upper Confidence Bound (UCB)** score. The specific formula used in `upperBound` is:

$$UCB = \underbrace{\frac{W + 0.5 \times D}{N}}_{\text{Exploitation}} + \underbrace{C \times \sqrt{\frac{\ln(N_{parent})}{N}}}_{\text{Exploration}} \tag{3.1}$$

By treating a Draw as half a Win (0.5), the agent is incentivized to avoid losses even if a win is not guaranteed.

Expansion

Once a leaf node is reached, if the game is not over, the algorithm expands the tree. The `expand` method:

1. Identifies all valid moves that have not yet been instantiated as children.
2. Selects one random unexplored move.
3. Generates the resulting board state.
4. Creates (or retrieves) the corresponding child node and links it to the parent.

Simulation (Rollout)

From the newly expanded node, the `simulate` method conducts a random playout.

- The algorithm plays random moves for both sides until a terminal state (Win/Loss/Draw) is reached.
- This phase requires no strategic knowledge, relying purely on the rules of the game to determine a result.
- This is also the bottle neck for MCTS effectiveness: if we do all random roll out, MCTS is weak.

Backpropagation

The result of the simulation is propagated up the tree using the `backprop` method.

- The algorithm traverses from the leaf back to the root using `node.parent` pointers.
- Statistics (N, W, D, L) are updated at every node along the path.

3.2.4 Action Selection and Symmetry Inversion

After the allocated iterations (default 2000) are complete, the agent selects the best move.

1. **Robust Child:** The algorithm selects the move with the highest visit count (N), as this represents the most thoroughly explored and statistically reliable path.
2. **Inverse Transformation:** Since the root node might have been transformed to a canonical state (e.g., rotated 90 degrees), the selected move must be mapped back to the original board orientation.

3.2.5 Conclusion

This implementation provides a robust, anytime algorithm for Tic-Tac-Toe. By leveraging canonical state hashing, it effectively solves the "state explosion" problem common in tree searches. The use of UCB with draw-weighting ensures the agent plays conservatively to avoid losses, making it a strong opponent even without the exhaustive calculation of Minimax. However, it doesn't perform well for the random roll out, especially if we increase the complexity by doing a 4 by 4 TicTacToe board

3.2.6 Pseudo code

Algorithm 3 Step 1: Selection

```
function Select(Node)
    while Node is fully expanded and has children do
        BestScore ← −∞
        BestChild ← None
        BestMove ← None
        for Move, Child ∈ Node.children do
            Exploit ← (Child.W + 0.5 × Child.D)/Child.N
            Explore ← C_exp × √(ln(Node.N)/Child.N)
            Score ← Exploit + Explore
            if Score > BestScore then
                BestScore ← Score
                BestChild ← Child
                BestMove ← Move
            end if
        end for
        Node ← BestChild
        Node.board ← Node.board.act(BestMove)
    end while
    return Node
end function
```

Algorithm 4 Step 2: Expansion

```
function Expand(Node, TranspositionTable)
    ValidMoves ← Node.board.get_valid_moves()
    Unexplored ← {m | m ∈ ValidMoves, m ∉ Node.children}
    if Unexplored is empty then return Node
    end if
    Move ← RandomChoice(Unexplored)
    NewBoard ← Node.board.act(Move)
    CanonCells, TransformID ← NewBoard.get_canonical()
    Key ← Hash(CanonCells)
    if Key ∈ TranspositionTable then
        Child ← TranspositionTable[Key]
    else
        Child ← NewNode(NewBoard, TransformID)
        TranspositionTable[Key] ← Child
    end if
    Child.parent ← Node
    Node.children[Move] ← Child
    return Child
end function
```

Algorithm 5 Step 3: Simulation

```
1: function Simulate(Board)
2:     Current ← Board.clone()
3:     while Current.isEnd() == Incomplete do
4:         Moves ← Current.get_valid_moves()
5:         RandomMove ← RandomChoice(Moves)
6:         Current ← Current.act(RandomMove)
7:     end while
8:     return Current.isEnd()                    ▷ Returns Win, Loss, or Draw
9: end function
```

Algorithm 6 Step 4: Backpropagation

```
1: function Backpropagate(Node, Result)
2:     while Node ≠ Null do
3:         Node.Visits ← Node.Visits + 1
4:         Player ← GetPlayerForDepth(Node.board.depth)
5:         if Result == Draw then
6:             Node.Draws ← Node.Draws + 1
7:         else if Result matches Player then
8:             Node.Wins ← Node.Wins + 1
9:         else
10:            Node.Losses ← Node.Losses + 1
11:        end if
12:        Node ← Node.parent
13:    end while
14: end function
```

Algorithm 7 MCTS Main Loop

Input: $RootBoard$: The current state of the game
Input: $Iterations$: Number of simulations to run
Output: $BestMove$: The optimal move index

```
1:  procedure Search(RootBoard, Iterations)
2:      TranspositionTable ← ∅
3:      Root ← GetOrCreateNode(RootBoard, TranspositionTable)
4:      for i ← 1 to Iterations do
5:          Leaf ← Root
6:          Leaf ← Select(Leaf)
7:          if Leaf.board.isEnd() == Incomplete then
8:              Leaf ← Expand(Leaf, TranspositionTable)
9:          end if
10:         Result ← Simulate(Leaf.board) Backpropagate(Leaf, Result)
11:     end for
12:     BestMove ← None
13:     MaxVisits ← −1
14:     for Move, Child ∈ Root.children do
15:         if Child.Visits > MaxVisits then
16:             MaxVisits ← Child.Visits
17:             BestMove ← Move
18:         end if
19:     end for
20:     if Root.canonical_transform ≠ 0 then
21:         BestMove ← InverseTransform(BestMove, Root.canonical_transform)
22:     end if
23:     return BestMove
24: end procedure
```

Chapter 4

Improved Solution

These strategies offer superior real-time inference speeds for Tic-Tac-Toe compared to conventional solvers, while some also demonstrating the scalability required to master Gomoku, a significantly more complex variant.

4.1 Q Learning

4.1.1 Q Learning Logic

The simplest form of learning agent: have a table with pre-train action-value pair for each board state and select the most rewarding move.

To solve the game effectively, we implemented a Reinforcement Learning approach based on Temporal Difference (TD) learning. Although often referred to as Q-Learning, our specific implementation focuses on learning the Value Function $V(s)$ for every possible board state s. This value represents the probability of the current player winning from that specific state.

Unlike complex games like Go or Chess, the state space of Tic-Tac-Toe is small enough to be fully enumerated. Before training begins, we generate all possible legal board configurations using a recursive Depth-First Search (DFS) approach.This is handled by the `getAllState` function, which starts from an empty board and recursively plays every valid move until a terminal state (Win, Loss, or Draw) is reached, either **exploit old state** (greedy) or **explore new state**. When game end, we update the all the states in the path that the agent take greedily by the equation.

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot [V(S_{t+1}) - V(S_t)] \tag{4.1}$$

Each unique state is hashed and stored in a dictionary, acting as our lookup table. This ensures that we have a memory slot allocated for every conceivable game situation.

4.1.2 Pseudo Code

Algorithm 8 Q Value Learning with Backpropagation

Input: α: Learning rate (Step size)
Input: ϵ: Exploration probability
Output: V: Optimized Value Table for all states

Phase 1: State Space Enumeration

$V \leftarrow \emptyset$
procedure GetAllStates($board$)
  $h \leftarrow$ Hash($board$)
  if $h \notin V$ **then**
    $V[h] \leftarrow$ InitValue($board$) ▷ 1.0 if Win, 0.0 if Loss, 0.5 otherwise
    for $move \in board.valid\_moves()$ **do**
      GetAllStates($board.act(move)$)
    end for
  end if
end procedure
GetAllStates(EmptyBoard)

Phase 2: Training (Self-Play)

for $epoch \leftarrow 1$ to N **do**
  $History \leftarrow []$
  $S_{curr} \leftarrow$ EmptyBoard
  while S_{curr} is not Terminal **do** ▷ Play Episode
    if Random() $< \epsilon$ **then**
      $a \leftarrow$ RandomMove()
      $greedy \leftarrow$ False
    else
      $a \leftarrow \text{argmax}_{a'} V(S_{curr}.\text{act}(a'))$
      $greedy \leftarrow$ True
    end if
    $S_{next} \leftarrow S_{curr}.\text{act}(a)$
    $History.\text{push}((S_{curr}, S_{next}, greedy))$
    $S_{curr} \leftarrow S_{next}$
  end while
  for $i \leftarrow$ Length($History$) down to 0 **do** ▷ Backpropagate Values
    $(S_t, S_{t+1}, greedy) \leftarrow History[i]$
    if $greedy$ is True **then** ▷ Only learn from rational moves
      $V(S_t) \leftarrow V(S_t) + \alpha \cdot [V(S_{t+1}) - V(S_t)]$
    end if
  end for
end for

4.2 Shared MCTS

A distinct feature of this implementation is the use of a **Shared Tree** during training. The `MCTSPlayer` class utilizes a class-level attribute `_shared_mcts`. This is based on how we wasted result of previous MCTS tree when building new one upon evaluating next state.

- **Mechanism:** Both the "Agent" and the "Opponent" (in self-play) read from and write to the

same memory structure.

- **Benefit:** This ensures that knowledge discovered by Player 1 is immediately available to Player 2. If Player 1 finds a winning strategy, Player 2 immediately perceives it as a losing path to be avoided, accelerating the discovery of the Nash Equilibrium.

4.2.1 Heuristic-Guided Search Phases

The core innovation in this algorithm lies in how it prioritizes moves. Instead of treating all legal moves as initially equal, the agent calculates a **Priority Score** based on immediate tactical threats.

In standard MCTS, the expansion step often selects a random unvisited child. Here, the `expand` method evaluates all potential moves and assigns them a priority score before selection:

Condition	Priority	Reasoning
Immediate Win	20	Terminate game favorably
Block Opponent Win	18	Prevent immediate loss
Create Threat	8	Setup potential fork
Center Square (4)	6	Strategic control
Corner Squares	4	High strategic value
Standard Move	0	Neutral

Table 4.1: Priority Scoring Logic in Expansion Phase

This priority queue ensures that the tree grows along critical paths (wins/blocks) first, mimicking human intuition.

4.2.2 Tactical Simulation (Heavy Playouts)

The `simulate` method replaces the traditional random rollout with a "tactically aware" simulation. During the playout, the agent does not choose moves uniformly at random. Instead, it checks for immediate wins or forced blocks at every step.

This drastically reduces the variance of the simulation result. A random rollout might miss a 1-move win and eventually lose, returning a noisy signal (0.0). A tactical rollout ensures that if a winning move exists, it is taken, returning an accurate signal (1.0).

4.2.3 Standard MCTS Components

Selection

Use the standard Upper Confidence Bound (UCB1) formula to balance exploration and exploitation:

$$Score = \underbrace{(1.0 - \mu_{child})}_{\text{Exploitation}} + \underbrace{C\sqrt{\frac{\ln(N_{parent})}{N_{child}}}}_{\text{Exploration}} \tag{4.2}$$

Where μ_{child} is the win-rate from the perspective of the child node's player. The code inverts this value $(1.0 - \mu)$ to view it from the parent's perspective.

Backpropagation

The `backpropagate` function updates the statistics of the traversal path. It assigns definite values based on the terminal state:

- **Win:** $+1.0$ (for the winning player's nodes)
- **Draw:** $+0.5$
- **Loss:** $+0.0$

4.2.4 Training Loop

The training process is governed by a self-play mechanism implemented in the `train` function.

1. **Temperature Decay:** Exploration is controlled via a temperature parameter. Early epochs use high temperature (0.1) to encourage diversity, while later epochs decay to 0.0 for exploitation.
2. **Alternating Starts:** The `Judger` swaps symbols (X/O) every epoch to ensure the agent learns to play both sides of the board evenly.
3. **Persistence:** The resulting tree is serialized using `pickle`, allowing the learned policy to be saved and reloaded for inference.

4.2.5 Conclusion

This implementation represents a significant improvement over naive MCTS for deterministic games. By encoding rule-based heuristics (Win/Block logic) directly into the search tree construction, the algorithm bypasses the "cold start" problem of Reinforcement Learning, allowing it to play at a competent level almost immediately while still refining its long-term strategy through MCTS iterations. However, this isn't an effective learning algorithm and it takes a long time to train a quality agent.

4.2.6 Pseudo Code

Algorithm 9 Priority Scoring Logic (Used in Expansion & Simulation)

```
function GetMovePriority(Board, Move, Player)
    Opponent ← Switch(Player)
    NextBoard ← Board.act(Move, Player)
    Priority ← 0
    if NextBoard.isEnd() == Player Wins then
        return 20                                              ▷ Immediate Win
    end if
                                                               ▷ Check Tactical Conditions
    if Move ∈ ImmediateWins(Board, Player) then
        Priority ← max(Priority, 20)
    else if Move ∈ ImmediateWins(Board, Opponent) then
        Priority ← max(Priority, 18)                           ▷ Forced Block
    else if Move ∈ Threats(Board, Player) then
        Priority ← max(Priority, 8)                            ▷ Creates Fork
    end if
                                                               ▷ Positional Heuristics
    if Move == Center then
        Priority ← max(Priority, 6)
    else if Move ∈ Corners then
        Priority ← max(Priority, 4)
    end if
    return Priority
end function
```

Algorithm 10 Heuristic Expansion (Best-First)

```
function Expand(Node, Board, Player)
    Unexplored ← {m | m ∈ Board.valid_moves(), m ∉ Node.children}
    if Unexplored = ∅ then return None
    end if
    Priorities ← List()
    for Move ∈ Unexplored do
        Score ← GetMovePriority(Board, Move, Player)
                                                               ▷ Add small random noise to break ties
        Priorities.add((Score + Noise, Move))
    end for
    Priorities.sortDescending()
    BestMove ← Priorities[0].Move
                                                               ▷ Create node only for the heuristically best move
    NewBoard ← Board.act(BestMove, Player)
    Child ← NewNode(NewBoard)
    Child.parent ← Node
    Node.children[BestMove] ← Child
    return Child
end function
```

Algorithm 11 Heuristic Simulation (Heavy Playout)

```
function Simulate(Board, StartPlayer)
    Curr ← Board.clone()
    Player ← StartPlayer
    Depth ← 0
    while Depth < 20 and Curr.isEnd() == Incomplete do
        ValidMoves ← Curr.valid_moves()
        if ValidMoves = ∅ then return Draw
        end if
        Priorities ← List()
        for Move ∈ ValidMoves do
            Score ← GetMovePriority(Curr, Move, Player)
            Priorities.add((Score, Move))
        end for
        BestMove ← ArgMax(Priorities)
        Curr ← Curr.act(BestMove, Player)
        Player ← Switch(Player)
        Depth ← Depth + 1
    end while
    return Curr.isEnd()
end function
```

Algorithm 12 Tactical Move Selection (Final Decision)

```
function GetBestMove(Node, Temperature)
    Board ← Node.board
    Player ← Node.player
    Opponent ← Switch(Player)
    if ∃m ∈ ImmediateWins(Board, Player) then
        return m                                   ▷ Always take the win
    end if
    if ∃m ∈ ImmediateWins(Board, Opponent) then
        return m                                   ▷ Always block the loss
    end if
    Moves ← Node.children.keys()
    Values ← List()
    for Move ∈ Moves do
        WinRate ← Node.children[Move].ValueSum/Node.children[Move].Visits
        Values.add(1.0 − WinRate)
    end for
    if Temperature == 0 then
        return ArgMax(Values)
    else
        NoisyValues ← Values + GaussianNoise(Temperature)
        return ArgMax(NoisyValues)
    end if
end function
```

4.3 Heuristic Alpha Beta Pruning

To address the expanded state space of Gomoku (often played on grids up to 15×15), we implemented an optimized variant of Alpha-Beta Pruning. Unlike the standard implementation used for Tic-Tac-Toe, this solution incorporates domain-specific heuristics to reduce the effective branching factor and a pattern-based evaluation function to assess non-terminal states.

4.3.1 Negamax Framework

The core search logic utilizes the **Negamax** algorithm, a simplified variant of Minimax that relies on the property $\max(a, b) = -\min(-a, -b)$ for zero-sum games. This allows the agent to use a single recursive function where the value of a state is defined relative to the current player, eliminating the need for separate maximizing and minimizing code blocks.

Since searching to the end of a Gomoku game is computationally infeasible, the algorithm relies on a static evaluation function to assess leaf nodes at a fixed depth (default 6). The evaluation function scans the board for specific stone patterns and assigns scores based on their strategic value:

- **Win Condition (5-in-a-row):** 10^9 points.
- **Open 4:** 10^8 points. This is a "forced win" state as the opponent cannot block both ends.
- **Closed 4 / Open 3:** 10^6 to 5×10^6 points. Strong threats that force opponent responses.
- **Minor Patterns:** Open 2s and isolated stones are given lower weights (10 to $5,000$) to guide positional play in the early game.

The final score is calculated as the weighted sum of the current player's patterns minus the weighted sum of the opponent's patterns.

4.3.2 Search Space Optimizations

To combat the high branching factor of Gomoku, the implementation utilizes two critical pruning techniques:

Dynamic Play Range (Search Window)

Standard Alpha-Beta checks every empty cell on the board. This implementation maintains a dynamic bounding box, the `PlayRange`, which tracks the minimum and maximum row/column indices of all placed stones.

- The search is strictly limited to this bounding box extended by a 2-cell margin.
- This ensures the agent only analyzes moves in the immediate vicinity of existing gameplay, ignoring the vast empty areas of the board.

Heuristic Move Ordering

The efficiency of Alpha-Beta pruning depends heavily on the order in which moves are evaluated. The `get_ordered_moves` function sorts potential moves to maximize cut-offs:

1. **Locality:** It filters moves to strictly fall within the active `PlayRange`.
2. **Adjacency:** It prioritizes moves that are immediate neighbors to existing stones.

By evaluating these high-probability moves first, the algorithm triggers Beta cut-offs earlier, allowing it to search deeper within the same time constraints.

4.3.3 Conclusion

The Heuristic Alpha-Beta algorithm serves as a critical first step toward mastering the complexity of Gomoku. By constraining the search space through dynamic windowing and aggressive pruning, this approach achieves the low latency required for real-time play, a significant improvement over naive search methods. However, its performance remains strictly bound by the quality of the static evaluation function.

4.3.4 Pseudo Code

Algorithm 13 Pattern-Based Board Evaluation

```
function Evaluate(Board, Player)
    Score_self ← 0,   Score_opp ← 0
    Patterns ← {Win, Open4, Closed4, Open3, ...}
    Weights ← {10^9, 10^8, 5 × 10^6, 10^6, ...}
    for every cell (r, c) and direction d do
        Pattern ← DetectConsecutive(Board, r, c, d)
        if Pattern ∈ Patterns then
            if Pattern.Owner == Player then
                Score_self ← Score_self + Weights[Pattern.Type]
            else
                Score_opp ← Score_opp + Weights[Pattern.Type]
            end if
        end if
    end for
    return Score_self − Score_opp
end function
```

Algorithm 14 Move Ordering with Dynamic Window

```
function GetOrderedMoves(Board, History)
                                                  ▷ 1. Define Search Window
    Range ← BoundingBox(History) + Margin(2)
                                                  ▷ 2. Filter Moves
    Candidates ← ∅
    for Move ∈ Board.ValidMoves do
        if Move is inside Range then
            Candidates.add(Move)
        end if
    end for
                                                  ▷ 3. Sort by Locality (Has Neighbor?)
    return SortByNeighborCountDescending(Candidates)
end function
```

Algorithm 15 Negamax Search

```
1: function Negamax(Board, Depth, α, β)
2:     Result ← Board.isEnd()
3:     if Result ≠ Incomplete then
4:         if Result is Draw then return 0
5:         end if
6:         return −(10^9 + Depth)                    ▷ Loss for current player
7:     end if
8:     if Depth == 0 then
9:         return Evaluate(Board, Board.CurrentPlayer)
10:    end if
11:    Moves ← GetOrderedMoves(Board, History)
12:    MaxEval ← −∞
13:    for Move ∈ Moves do
14:        Board.MakeMove(Move)
15:        Score ← −Negamax(Board, Depth-1, -β, −α)
16:            Board.UndoMove()
17:            MaxEval ← max(MaxEval, Score)
18:            α ← max(α, Score)
19:            if α ≥ β then
20:                break
21:            end if
22:        end for
23:        return MaxEval
24:    end function
```

4.4 Temporal-Difference Learning

To overcome the scalability limitations of tabular methods, we implemented a Reinforcement Learning agent based on **Temporal-Difference (TD) Learning** with **Linear Value Function Approximation**. Instead of storing a value for every unique state, this approach approximates the value of a board state $V(s)$ as a linear combination of features and weights. This allows the agent to generalize across similar board configurations and significantly reduces memory requirements.

4.4.1 Value Function Approximation

The core of the agent is the linear evaluation function:

$$V(s) \approx W^T \cdot X(s) = \sum_{i=1}^{n} w_i \cdot x_i(s) \tag{4.3}$$

Where:

- W is a learnable weight vector.
- $X(s)$ is a feature vector extracted from the board state s.
- $V(s)$ represents the expected utility of the state.

4.4.2 Feature Extraction

The feature vector $X(s)$ is constructed by analyzing all possible winning lines (rows, columns, and diagonals) on the board. The extract feature quantifies the "potential" of each sequence:

- **Positive Features**: Assigned to sequences favorable to the agent (e.g., containing only the agent's pieces). The magnitude is scaled by the number of pieces present ($Count/WIN\_LENGTH$).
- **Negative Features**: Assigned to sequences favorable to the opponent.
- **Zero**: Assigned to blocked or empty sequences.

This encoding captures the strategic importance of board patterns without requiring an exhaustive state lookup.

4.4.3 LMS Update

The agent updates its weight vector W using the **Least Mean Squares** (LMS) rule, derived from the gradient of the squared error between the estimated value and a target value.

The update rule applied to each step t is:

$$W \leftarrow W + \alpha \cdot (V_{target}(s_t) - V_{est}(s_t)) \cdot X(s_t) \tag{4.4}$$

Where the target value $V_{target}(s_t)$ is determined via bootstrapping:

- **Intermediate States**: $V_{target}(s_t) = 0 + \gamma \cdot V_{est}(s_{t+1})$ (where γ is the discount factor).
- **Terminal States**: $V_{target}(s_{final}) = Reward$ (where Reward is $+1$ for Win, -1 for Loss, 0 for Draw).

This allows the agent to propagate rewards backward from the end of the game to earlier states, learning which features contribute most to victory.

4.4.4 Training Procedure

Training is conducted via self-play between two identical TD agents. The process follows a structured exploration schedule:

- Policy: The agents utilize an ϵ-greedy strategy, choosing the move that maximizes $V(s_{next})$ with probability $1 - \epsilon$, and a random move with probability ϵ.
- Exploration Decay: To facilitate convergence, the exploration rate ϵ decays exponentially after each game (e.g., $\epsilon \leftarrow \max(\epsilon_{min}, \epsilon \cdot 0.995)$).

This ensures the agents initially explore diverse strategies before stabilizing into optimal play.

4.4.5 Conclusion

The Temporal-Difference Learning agent with linear value function approximation proves to be a highly effective and efficient solution for Tic-Tac-Toe. By compressing the state space into a compact feature vector, the algorithm avoids the computational overhead of tabular methods, allowing for rapid training convergence and low-latency inference.

However, the current implementation relies on a greedy policy, selecting moves based solely on a 1-step lookahead of value estimates. To further elevate performance, particularly for complex variants like Gomoku, this learned value function can be integrated into the **Heuristic Alpha-Beta Pruning** framework described previously. By replacing the static, hand-crafted evaluation function with the learned weights ($V(s)$) at the leaf nodes of a deep search tree, the system can move beyond simple greedy selection.

4.4.6 Pseudo Code

Algorithm 16 Linear Feature Extraction

```
1: function ExtractFeatures(Board)
2:     Lines ← Board.getAllLines()                          ▷ Rows, Cols, Diagonals
3:     Features ← Vector of zeros, length |Lines|
4:     for i ← 0 to |Lines| − 1 do
5:         Sum ← Count(Lines[i])                            ▷ X=+1, O=-1 per cell
6:         if Sum > 0 then                                  ▷ Line favors X (no O's present)
7:             Features[i] ← Sum/WIN_LENGTH
8:         else if Sum < 0 then                             ▷ Line favors O (no X's present)
9:             Features[i] ← Sum/WIN_LENGTH
10:        else
11:            Features[i] ← 0                              ▷ Blocked or Empty
12:        end if
13:    end for
14:    return Features
15: end function
```

Algorithm 17 Value Function Approximation

```
1: function SelectAction(Board, Weights, ϵ)
2:     if Random() < ϵ then
3:         return RandomMove(Board)
4:     end if
5:     BestVal ← −∞
6:     BestMove ← None
7:     for Move ∈ Board.ValidMoves do
8:         NextState ← Board.Act(Move)
9:         Val ← GetValue(NextState, Weights)
10:        if Val > BestVal then
11:            BestVal ← Val
12:            BestMove ← Move
13:        end if
14:    end for
15:    return BestMove
16: end function
```

Algorithm 18 TD Training with LMS Update

procedure TrainAgents($Episodes, \alpha, \gamma$)
  for $e \leftarrow 1$ **to** $Episodes$ **do**
    $History \leftarrow []$
    $Board \leftarrow$ EmptyBoard
    while $Board$ is not Terminal **do**
      $X_t \leftarrow$ ExtractFeatures($Board$)
      $History.push(X_t)$
      $Move \leftarrow$ SelectAction($Board, W, \epsilon$)
      $Board \leftarrow Board.Act(Move)$
    end while
    $Reward \leftarrow$ GetReward($Board$)
    for $t \leftarrow 0$ **to** $|History| - 1$ **do**
      $X_t \leftarrow History[t]$
      $V_{est} \leftarrow W \cdot X_t$
      if $t == |History| - 1$ **then**
        $V_{target} \leftarrow Reward$
      else
        $X_{next} \leftarrow History[t+1]$
        $V_{target} \leftarrow 0 + \gamma \cdot (W \cdot X_{next})$ ▷ Bootstrap
      end if
      $Error \leftarrow V_{target} - V_{est}$
      $W \leftarrow W + \alpha \cdot Error \cdot X_t$
    end for
    $\epsilon \leftarrow \epsilon \cdot$ DecayRate
  end for
end procedure

4.5 Alpha Zero Deep Learning Model

To achieve better performance without relying on human expert data, we implemented an **AlphaZero-style** Deep Reinforcement Learning architecture. This approach represents a paradigm shift from the heuristic methods described previously. Instead of hard-coded evaluation functions, the agent learns a complex strategy entirely through self-play, utilizing a deep neural network to guide a Monte Carlo Tree Search (MCTS).

4.5.1 Unified Neural Network Architecture

At the core of the system is a dual-headed Convolutional Neural Network (CNN), referenced as `SimpleCNN` in the codebase. The choice for CNN instead of an MLP is due to spatial invariance feature of CNN. In a Gomoku board, some exact plays could be anywhere on the board, and to save resources, a CNN which slide through the board is most fitted for this calculation. Unlike the original AlphaGo which used separate networks for policy and value, this implementation uses a single network $f_\theta(s)$ that takes the board state as input and produces two simultaneous outputs:

- **Policy Head (p):** A probability distribution over all possible moves ($P(a|s)$), acting as a "prior" intuition for the search.
- **Value Head (v):** A scalar evaluation in the range $[-1, 1]$, estimating the probability of winning from state s.

The input to the network is a spatial tensor of shape $(3 \times H \times W)$, encoding the current player's stones, the opponent's stones, and empty intersections as separate channels.

4.5.2 Neural Monte Carlo Tree Search (MCTS)

The search mechanism, implemented in the `NeuralMCTS` class, modifies the standard MCTS selection phase to incorporate the neural network's outputs.

Guided Selection (PUCT)

Instead of relying solely on visit counts, the selection of child nodes is driven by the **Predictor + UCT (PUCT)** formula. For a node s and action a, the selection score is calculated as:

$$U(s,a) = Q(s,a) + C_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{4.5}$$

Where $Q(s,a)$ is the mean value of the action, and $P(s,a)$ is the prior probability output by the policy network. This allows the search to prioritize moves the network deems "promising" while maintaining exploration.

Evaluation instead of Rollout

A critical optimization in this architecture is the removal of random simulations (rollouts). When the search reaches a leaf node, it does not play to the end of the game randomly. Instead, it queries the Value Head of the network to obtain an immediate estimate v, which is then backpropagated up the tree.

4.5.3 Self-Play Training Pipeline

The training process is cyclical and entirely autonomous, managed by the `train_alphazero_style` function:

1. **Data Generation:** The agent plays games against itself. At each step, it executes an MCTS search to determine the best move. The visit counts of the search tree (π) are stored as the "ground truth" policy target, effectively using MCTS as a policy improvement operator.
2. **Optimization:** The network is trained on the collected self-play data to minimize a joint loss function:
$$L = (z - v)^2 - \boldsymbol{\pi}^T \log \mathbf{p} + c||\theta||^2 \tag{4.6}$$
where $(z - v)^2$ is the Mean Squared Error between the predicted value and the actual game result, and $-\boldsymbol{\pi}^T \log \mathbf{p}$ is the Cross-Entropy loss between the network's policy and the MCTS search probabilities.
3. **Canonical Symmetries:** To maximize data efficiency, the board states are transformed (rotated and mirrored) into a canonical form before neural inference and storage.

4.5.4 Conclusion

This architecture allows the agent to bootstrap its knowledge from zero. By iteratively training the neural network to predict the results of its own MCTS search, the system creates a positive feedback loop: the network improves the search, and the search generates better data for the network.

4.5.5 Pseudo Code

This part won't have pseudo code due to the complexity of the method. Though you can read more at the source code listed in the abstract.

Chapter 5

Evaluation of different solutions

To rigorously assess the performance of our implemented agents, we conducted a series of match-ups against established baselines. The evaluation criteria focus on two key metrics: **Win Rate**, representing the strategic strength of the model, and **Average Inference Time**, representing the computational efficiency suitable for real-time play.

5.1 3-by-3 Tic-Tac-Toe

In the 3x3 domain, the state space is small enough (3^9) to be solved completely. Therefore, we evaluate all learning agents against a **Perfect Alpha-Beta Pruning** agent. These are result of 100 games each:

Method	Win Rate	Loss Rate	Draw Rate	Avg Time
Q Learning	0.0%	50.0%	50.0%	0.70 s
TD Learning (Linear)	0.0%	50.0%	50.0%	0.94 s
MCTS (Shared)	0.0%	50.0%	50.0%	0.85 s
MCTS (Base)	1.0%	99.0%	0.0%	2.53 s
Heuristic (depth 10)	0.0%	0.0%	100.0%	2.66 s

Table 5.1: Performance against Perfect Alpha-Beta Player (3x3)

Analysis:

- **TD Learning, MCTS shared abd Q table** demonstrates superior speed and great strength against perfect player.
- **MCTS** has a abnormality in winning a game against perfect player, this can only be assumed because of the randomness in MCTS making plays that even perfect search fail to check, since we pruning path if it doesn't meet value before.

5.2 5-by-5 Tic-Tac-Toe

As the board size increases to 5x5, the state space expands significantly, rendering perfect Minimax intractable. For this evaluation, the opponent is a **Heuristic Alpha-Beta agent searching at Depth 4** for 10 games.

Method	Win Rate	Loss	Draw Rate	Avg Time
TD Learning (Linear)	100.0%	0.0%	0.0%	14.79 s
MCTS (Shared Tree)	100.0%	.0%	0.0%	1043.48 s
AlphaZero	**2**0%	20.0%	60.0%	38.06 s

Table 5.2: Performance against Heuristic Alpha-Beta Depth 4 (5x5)

Analysis:

- **MCTS shared**: taking much longer to play, and the quality isn't good enough for the time it was trained for. Showing that if we expand MCTS to larger board, it can't scale.
- **AlphaZero**: playing well, although was trained too little, making evaluation not great.

5.3 10-by-10 Gomoku

This is the most complex domain evaluated. The opponent remains the **Heuristic Alpha-Beta (Depth 3)**, but the branching factor is now massive (≈ 100). Pure MCTS shared fails to converge within reasonable time limits here, so we focus on the comparison between the lightweight TD agent and the Deep Learning approach. All results are of 10 games.

Method	Win Rate	Loss	Draw Rate	Avg Time (ms)
TD Learning (Linear)	60.0%	40.0%	0.0%	26.70 s
AlphaZero (800 sims)	0.0%	100.0%	0.0%	27.82 s

Table 5.3: Performance against Heuristic Alpha-Beta Depth 3 (10x10)

Analysis:

- **TD Learing**: still performing great, even with little traning, and move fast.
- **AlphaZero**: badly trained, due to taking too much time, though making move fast.

5.4 Training time

Method	5x5	10x10
TD Learning (Linear) (100 game)	2 m	10 m
AlphaZero (100 sims) (5 epochs)	10 m	253 m

Table 5.4: Training time for the methods

Analysis:

- **TD Learing**: training quickly and perform well with little resource.
- **AlphaZero**: it seems the model to be too complex, taken a long time to train and isn't playing perfectly.

Chapter 6

Discussion

While the current agents demonstrate competent gameplay, several avenues for improvement remain to elevate their performance to a competitive or superhuman level. Future work could focus on the following areas:

- **Architectural Improvements for AlphaZero:** The current implementation utilizes a 'SimpleCNN'. To capture the complex long-term dependencies of Gomoku, the network should be upgraded architecture.
- **Advanced Heuristic Functions:** The heuristic evaluation used in the Alpha-Beta agent relies on static weights for fixed patterns. A more robust approach would be to research dynamic pattern recognition or implement **Victory by Continuous Threats (VCT)** algorithms
- **Hybrid Approaches:** Combining the speed of the Temporal-Difference (TD) agent with the lookahead capability of MCTS could yield a high-performance, low-latency bot.

Chapter 7

Conclusion

The primary objective of this project was to explore and catalog the spectrum of possible solutions for Tic-Tac-Toe and its complex variant, Gomoku. Rather than focusing narrowly on a single method, the project aimed for breadth, successfully implementing and comparing a wide range of Artificial Intelligence paradigms—from classical algorithmic searches (Minimax, Alpha-Beta Pruning) to stochastic simulations (MCTS) and modern Deep Reinforcement Learning (AlphaZero).

For small state spaces like 3×3 Tic-Tac-Toe, exhaustive search methods such as **Minimax** and **Alpha-Beta Pruning** remain the gold standard, providing mathematically optimal play with negligible computational cost. While learning agents like **Temporal-Difference (TD) Learning** and **AlphaZero** successfully converged to optimal strategies, the training overhead offers no practical advantage in such solvable environments.

However, as complexity scaled to 10×10 Gomoku, the limitations of brute-force search became evident. The exponential growth of the game tree rendered Minimax intractable, and even Heuristic Alpha-Beta Pruning suffered from the "horizon effect" due to depth limits. In this domain, the **AlphaZero** architecture demonstrated superior capability. By substituting exhaustive rollout with a learned evaluation function, it successfully balanced the trade-off between search depth and intuition, proving that deep reinforcement learning is the necessary evolution for mastering complex, high-dimensional decision spaces where traditional search falters.

Although there are numerous aspects of the implementations that could be optimized, this project served as a comprehensive "warm-up" to the field of AI. It provided practical exposure to the distinct challenges of different branches: the computational constraints of exhaustive search, the convergence issues of tabular Reinforcement Learning, and the data efficiency problems of Deep Learning.

References

Kalra, B. (2022), Generalized agent for solving higher board states of tic tac toe using reinforcement learning, *in* '2022 Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC)', pp. 715–720.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K. and Hassabis, D. (2017), 'Mastering chess and shogi by self-play with a general reinforcement learning algorithm', *ArXiv* **abs/1712.01815**. **URL:** *https://api.semanticscholar.org/CorpusID:33081038*