Reinforcement Learning for ConnectX

Tejashvi Singh (PH-45)
Btech CSE
MIT WPU
Pune, India
1032212055@mitwpu.edu.in

Palak Gupta (PH-44)
Btech CSE
MIT WPU
Pune, India
1032212017@mitwpu.edu.in

Shefali Jaiswal (PH-54)
Btech CSE
MIT WPU
Pune, India
1032212770@mitwpu.edu.in

Aarya Jagdale (PH-64)
Btech CSE
MIT WPU
Pune, India
1032221518@mitwpu.edu.in

Alok Raj (Ph-13)
Btech CSE
MIT WPU
Pune, India
1032210741@mitwpu.edu.in

Prof Ruhi Patankar
Assistant Professor
School of CET
MITWPU PUNE

Prof Sarika Borade
Professor
School of CET
MITWPU PUNE

Abstract— In the realm of strategic board games lies ConnectX, a captivating rendition akin to the classic Connect 4. In this engaging contest, two adversaries strategize to align a sequence of X tokens either horizontally, vertically, or diagonally upon a customizable board of varying dimensions. The triumphant player is the one who deftly orchestrates this arrangement before their opponent. Our endeavors delve into the realm of artificial intelligence, delving deep into the arsenal of reinforcement learning methodologies. Within this manuscript, we meticulously explore and deploy a plethora of reinforcement algorithms in the pursuit of mastering the art of ConnectX gameplay.

I. Introduction

ConnectX extends this idea, allowing different board sizes and the number of tokens needed to win. This makes it more challenging, as there's no guaranteed winning strategy beyond certain limits. To tackle this, we applied various algorithms like Carlo tree search, Alpha Zero, Min-Max, and Greedy algorithms on Kaggle. These algorithms competed against each other in a Kaggle competition. AlphaZero performed the best, earning us a 9th place position out of 225 teams.

II. Related Work

A. History

We couldn't find any previous work specifically about using Reinforcement Learning to play ConnectX. However, Connect 4 has been extensively studied. It was first solved by James Dow Allen. Some detailed strategies for Connect 4 have been providedin previous research.

B. In Addition,

Connect 4 has been brute-force solved using methods like an 8-ply database and Mini Max & transposition tables. (MCTS) is suitable for problems where good actions aren't known in advance. It was combined with neural networks in AlphaGo, which excelled in Go. This led to the creation of AlphaZero, a more general version. Some variants focus on proven wins/losses, especially in sudden-death games. MCTS has been successfully applied to various games beyond Go.

Defining The Problem

In the game environment, players are tasked with developing a function named "player" on an M×N board. This involves strategic decision-making within a constrained time frame of 5 seconds per move. Failure to select a move within this timeframe results in an automatic loss. Additionally, making an illegal move, such as selecting a column that is already full, leads to an immediate loss. Therefore, the primary challenge lies in devising an efficient algorithm within the time constraints that maximizes the chances of winning and ensures compliance with the game's rules to avoid premature defeat. Methodology

A. The Greedy algorithm

The Greedy algorithm is known for its simplicity and effectiveness in solving specific problems by making the best choice at each step. It's widely used in fields like artificial intelligence and operations research, especially in games and networking. Games like ConnectX, focus on immediate gains without considering long-term consequences, making them useful for quick decisions.

In networking, it helps optimize data flow by prioritizing paths with the highest throughput or least congestion. However, it has limitations, such as settling for local optimal solutions that may not be the best overall.

1) Greedy Algorithm (Alpha-Beta pruning)

Minimax is a traditional strategy used in games like Connect X. It looks ahead at possible moves for both players, assuming the opponent will play optimally to minimize the player's score. It keeps searching until it reaches an end state and then returns a final score. By maximizing its score while minimizing the opponent's, Minimax aims for the best possible outcome. Algorithm 2, a version with alpha-beta pruning, was implemented. It uses a heuristic to evaluate the game state, assigning values based on the number of tokens in a row.

Algorithm 1

Require: *board, mark*
  $value_i \leftarrow 0 \; \forall i \in \{0, 1, ..., N-1\}$
  for $i \in \{0, 1, ..., N-1\}$ **do**
    if placing *mark* in column *i* connects *X* in a row for *mark* **then**
      $value_i \leftarrow \infty$
    end if
    return $a = arg\,max_i\, value_i$
  end for

formulation is:

$$UCB(\text{Tree rooted at state } S) = \frac{w_S}{n_S} + c \cdot \sqrt{\frac{\ln N_S}{n_S}}$$

where :

w_S = The number of winning simulations from state S

n_S = The total number of simulations from state S

N_s = The total number of simulations from the parent of state

```
Require: board, mark, heuristic
  function MINIMAX(state, depth, α, β, maximizingPlayer)
    if depth = 0 or game ended then
      return heuristic(state, mark)
    end if
    if maximizingPlayer then
      value ← −∞
      for each legal move do
        state' ← state reached by playing this move
        value ← max(value, MINIMAX(state , 'depth − 1, α, β, FALSE)
        if value ≥ β then
          break
        end if
        α = max(α, value)
      end for
    else
      value ← ∞
      for each legal move do
        state' ← state reached by playing this move
        value ← min(value, MINIMAX(state', depth − 1, α, β, TRUE)
        if value ≤ α then
          break
        end if
        β = min(β, value)
      end for
    end if
  end function
  MINIMAX(board, 5, −∞, ∞, TRUE)
```

B. Plain Monte Carlo Tree Search

It creates a tree of potential game states from the current one, then iteratively follows four steps: selecting moves based on a balance of trying new options and picking those with high win chances, expanding the tree by adding promising untried moves, simulating random playthroughs from newly added nodes to estimate their win probabilities, and updating the tree based on the results of these simulations. This process helps MCTS focus on promising moves while still exploring less travelled paths. In each round of MCTS, moves are chosen using an upper confidence bound-like method to represent the value of different move options.

1) Alpha Zero

The AlphaZero algorithm employs a multifaceted approach to master games through self-play. Initially, MCTS navigates the game tree, systematically evaluating potential moves. This exploration process prioritizes actions with the highest Upper Confidence Bounds (UCB) values, culminate in a probability distribution over possible moves.

Upon reaching a terminal state in the game, outcomes are determined based on predefined rules, delineating victories, defeats, or draws. Subsequently, a neural network is engaged to map the current game state to two essential components: the policy, representing move probabilities, and the value function, indicating the likelihood of winning from that position.

This neural network undergoes refinement through backpropagation facilitated by the Adam optimizer. This iterative process aims to minimize errors in value predictions and discrepancies between the policy vector and the probabilities derived from the search process.

To optimize learning efficiency, AlphaZero incorporates experience replay, enabling the algorithm to iteratively refine its strategies with a growing corpus of training data while operating on reduced simulation data for each iteration. This integrated framework enables AlphaZero to progressively enhance its gameplay prowess through self-improvement and iterative learning.

Algorithm 3 Policy Iteration through Self-Play

Require: *executed episode, trainNN, initNN, threshold*
  function POLICY_ITERATION(numIters, numEpisodes)
    $\vartheta \leftarrow initNN()$
    trainExamples ← []
    for *i in* {1, ..., *numIters*} **do**
      for *e in* {1, ..., *numEpisodes*} **do**
        ex ← *executeEpisode*(*nn*)
        trainExamples.append(*ex*)
      end for
      $\vartheta_{new} \leftarrow trainNN(trainExampples)$
      if percentage(ϑ_{new} beats ϑ) ≥ *threshold* **then**
        $\vartheta \leftarrow \vartheta_{new}$
      end if
    end for
  end function

Algorithm 4 Execute Episode

Require: *assignReward, MCTS, gameNextState, gameStartState*
  function EXECUTEEPISODE(numSims)
    examples ← []
    s ← *gameStartState*()
    while True **do**
      for *i in* [1, ..., *numSims*] **do**
        MCTS(*s*, ϑ)
      end for
      examples.add((*s*, π_s, _))
      $a_* \sim \pi_s$
      s ← *gameNextState*(*s*, a_*)
      if *gameEnded*(*s*) **then**
        //Fill _ with rewards in examples, the backpropagation step for MCTS
        examples ← *assignReward*(*examples*)
        break and return *examples*
      end if
    end while
  end function

The Neural Network

Neural networks are used to learn game strategies in reinforcement learning. They predict the best move based on a given game state, with layers that process the data and an output layer predicting move probabilities. Training involves creating a dataset of game experiences, either through self-play or human-played games and then training the network on this data to associate board configurations with winning moves. Unlike MCTS and Minimax, neural networks don't explore the game tree but rely on learned patterns.

A) Brief Breakdown of the Neural Network

a) Initialize Parameters: Set up the network architecture and initialize weights and biases.

b) Forward Propagation: Process input data through the network to generate predictions.

c) Backward Propagation (Backpropagation): Adjust weights and biases based on the computed loss to improve predictions.

d) Repeat: Iterate through forward and backward propagation for a fixed number of times or until the loss reaches an acceptable level.

e) Evaluate and Tune: Assess performance on a separate validation set and adjust parameters as needed to improve performance and prevent overfitting.

f) Testing: Test the model's performance on unseen data to ensure it generalizes well.

g) Deployment: Deploy the trained model for use in applications or further analysis.

III. Observation and Comparisons

A. Monte Carlo Tree Search (MCTS)

a) Role: MCTS acts as a heuristic search algorithm within the RL framework. It efficiently navigates the vast game space of Connect X, balancing exploration (trying new moves) and exploitation (focusing on winning moves) through its iterative selection and simulation process.

Comparison:

- Advantages:
  - Efficient exploration with good winrates.
  - Relatively fast execution due to focusing on promising branches.
  - Adaptable to different game variations (board size, number of tokens to connect).
- Disadvantages:
  - Performance can deteriorate in games with massive branching factors.
  - Requires careful selection policy design for optimal balance between exploration and exploitation.

B. Minimax Algorithm

a) Role: Minimax examines all possible future moves and outcomes, guaranteeing the best move selection against an equally perfect opponent.

Comparison:

1) Advantages:
   - Provides the optimal move selection for perfect-information games with deterministic outcomes.
   - Relatively simple to understand and implement.
2) Disadvantages:
   - Computational cost explodes with increasing game complexity (large boards, many tokens to connect).
   - Impractical for real-time decision-making due to the extensive computations required.
   - Assumes perfect knowledge of the opponent's strategy, which may not hold in real-world scenarios.

C. Neural Networks

a) Role: Neural networks provide a data-driven approach to learning optimal game-playing strategies. They learn complex patterns from training data, allowing them to identify winning moves and adapt to different game variations.

Comparison:

1) Advantages:
   - Can learn complex strategies from large datasets of game experiences.
   - Can potentially generalize well to unseen game situations.

CONCLUSION

According to Table 1 & Table 2, plain MCTS performed better when compared to Min-Max. We conclude that when algorithms are compared some algorithms work better than others under some conditions.

Table 2:

Algorithms	Score
Greedy	260
Plain MCTS	1050
Minimax	873
MCTS-minimax hybrid	980
AlphaZero	**1282**

A. Kaggle Score

In our study, we compared different algorithms by making them play against each other, which helped us rank them. Our results from the Kaggle competition confirmed our findings and gave us a standard to measure the agents.

1) Looking ahead

Basic neural network architecture was used and the hyperparameters weren't finely tuned. With more time and computing power, we could fine-tune these aspects to further boost AlphaZero's performance.

ACKNOWLEDGEMENT

WE WOULD LIKE TO THANK PROF. RUHI PATANKAR FOR HIS ASSISTANCE THROUGHOUT THE PROJECT AND MAKING US FAMILIAR WITH THE CONCEPTS OF REINFORCEMENT LEARNING.

REFERENCES

[1] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

[2] Russell, S. J., & Norvig, P. (2021). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.

[3] Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press.

[4] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Tavener, S. (2012). A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games, 4(1), 1-43.

[5] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359.

[6] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

[7] Hassabis, D., Kumaran, D., Summerfield, C., & Botvinick, M. (2017). Neuroscience-inspired artificial intelligence. Neuron, 95(2), 24