Connect4 using Reinforcement Learning

Arvind Raghavendran (MDS202214)
Sarvesh Bhandary (MDS202240)
Sneha KK (MDS202254)

May 16, 2024

Contents

1 Introduction

Reinforcement Learning (RL) has emerged as a powerful paradigm for teaching agents to make decisions in various environments, including games. By interacting with an environment and receiving feedback in the form of rewards, RL algorithms learn to maximize cumulative rewards over time. This approach has been successfully applied to games due to their well-defined rules and clear objectives. In this project, we explore the application of RL techniques to Connect4, a classic two-player game known for its simple rules yet complex strategic depth. Connect4 offers an ideal environment for RL experimentation due to its discrete action space and deterministic state transitions.

2 Connect4 - The Game

Connect4 is a two-player game played on a 6x7 grid, where players take turns dropping colored discs into the columns. The objective is to connect four of one's own discs vertically, horizontally, or diagonally before the opponent does. The state space consists of all possible configurations of the game board, while the action space consists of the seven columns where a player can drop a disc.

The game's outcome is determined by three conditions:

- **Win (W)**: A player wins if they successfully connect four of their discs.
- **Loss (L)**: A player loses if their opponent connects four discs first.
- **Draw (D)**: The game ends in a draw if the entire grid is filled without either player achieving a win.

Additionally, we will utilize the Connect4 environment provided by Kaggle, which offers a convenient interface for interacting with the game environment and evaluating RL algorithms.

2.1 Kaggle Connect4 Environment Overview

The Kaggle Connect4 environment provides a comprehensive platform for implementing and testing RL algorithms in the context of the Connect4 game. It offers various features and functionalities that facilitate experimentation and evaluation.

2.2 Features

The environment includes the following key features:

- **Game Representation**: The game state is represented as a 6x7 grid, where each cell can be empty or occupied by a disc belonging to one of the players.

- **Action Space**: Players can choose actions by specifying the column in which they want to drop their disc. The action space consists of seven possible columns.
- **Game Logic**: The environment enforces the rules of Connect4, ensuring that players' actions are legal and updating the game state accordingly.
- **Reward System**: Rewards are provided based on the game outcome. A positive reward is given for winning, a negative reward for losing, and a neutral reward for a draw.

2.3 Functionalities

The environment offers several functionalities to support RL experimentation:

- **Game Initialization**: It allows for the initialization of new game instances, enabling researchers to start fresh games for each experiment.
- **Action Execution**: Players can execute actions by specifying the column in which they want to drop their disc. The environment handles the execution of actions and updates the game state accordingly.
- **State Observation**: Researchers can observe the current state of the game, including the positions of discs on the grid, to inform their decision-making process.
- **Reward Calculation**: Rewards are calculated based on the game outcome and provided to the agent at the end of each game episode, facilitating the learning process.

Overall, the Kaggle Connect4 environment offers a convenient and versatile platform for developing and evaluating RL algorithms in the context of the Connect4 game. Its features and functionalities streamline the experimentation process and enable researchers to focus on algorithm development and analysis.

3 Prior Knowledge

Connect4 presents an interesting challenge due to its relatively large state space and branching factor. The state space complexity arises from the number of possible configurations of the game board, which grows exponentially as the game progresses. Despite its apparent simplicity, Connect4 has been the subject of various studies and research efforts aimed at exploring optimal strategies and solving the game. In this section, we review existing literature on Connect4, focusing on approaches to address the state space complexity and insights gained from previous studies.

4 Initial Experiments

We shall investigate the efficacy of traditional reinforcement learning methods in the context of the game Connect4. Approaches such as value iteration, which entail iterating over the entire state space, are deemed impractical due to the sheer size of the state space, which comprises trillions of states. Consequently, convergence to state values is unattainable within a reasonable computational timeframe. Hence, Monte Carlo techniques like SARSA(eq 4.1.1) and Q-learning are employed. These methods update based on states, actions, and rewards observed in sampled episodes. An epsilon-greedy policy with an epsilon value of 0.2 and a learning rate of 0.1 is adopted. Should the agent encounter a previously unobserved state during training, it resorts to making a random move.
4.1.1 SARSA update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

4.1.2 Qlearning update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The reward distribution is defined as follows: a win yields a reward of +1, a loss incurs a reward of -1, a draw results in a reward of -0.5, and attempting an invalid action (i.e., placing a coin in an already filled column) warrants a significantly negative reward of -2. This penalty for invalid actions is crucial to discourage the agent from resorting to such moves, particularly when facing imminent defeat. Intermediate states within gameplay do not receive rewards, precluding the utilization of experience replay techniques that rely on sampling states, actions, and rewards within episodes. To train the models, random and negamax opponents are considered. While training against a proficient adversary like negamax is expected to imbue the agent with the ability to play competitively, this comes at the expense of increased training time, as a random opponent is considerably faster. Three agents are trained: SARSA with random for 1 million episodes, Q-learning with random for 1 million episodes, and Q-learning with negamax for a similar duration but fewer episodes (10,000). Subsequently, these models are evaluated against a random player, with the results presented in table 1.

Table 1: Performance of traditional models vs random agent in 1000 games

Agent(training method)	**Wins**	**Invalid moves/draws**	**losses**
SARSA with random	442	347	211
Qlearning with random	597	238	165
Qlearning with negamax	446	243	211

The analysis of the results reveals that Q-learning with a random opponent yields the most favorable outcomes. This observation highlights not only the superiority of Q-learning over SARSA but also the significant time overhead

associated with training against negamax, rendering it less practical.Our agents are also not very good in avoiding invalid moves.
Even though our models are able to win against a random agent, they have seen very less during training. During 1 lakh episodes of training , since we can atmost see only 42 states in an episode, we will be able to visit only a maximum of 42 lakh states. As depicted in Figure 1, an estimate of the distinct states visited versus the number of training episodes illustrates that we are able to explore approximately 500,000 states within 100,000 episodes. This accounts for less than 0.00001 percentage of the entire state space, indicating minimal learning despite extensive training. This explains the ineffectiveness of traditional RL methods when confronted with a vast state space.

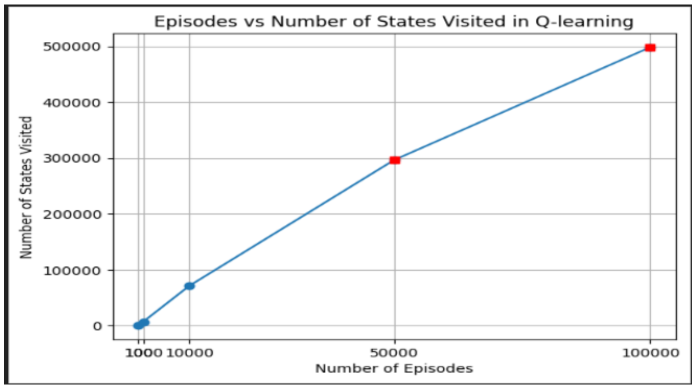


Figure 1

5 Deep Q-Networks

The Deep Q-Networks (DQNs) try to tackle the problem of the large state space faced by our traditional RL algorithms due to their limitations by design. The major problem faced was that even after running a large number of episodes, only a very tiny fraction of the entire state space could be explored, and yet, all the information gathered from the visited states was completely nontransferable to other states. That is, for instance, we might have observed states where our opponent had 3 tokens in a row multiple times and potentially lost in such situations, but even if the configuration of the grid is slightly different such that we have the same situation (3 tokens in a row for the opponent) but this exact state had not been observed before, the algorithm would not be able to utilize any past information and would end up taking a random action.
DQNs tackle this issue by extracting optimal features out of the grid and learn to move based on these features. The architecture is also such that the weights are shared across all states, so 2 states with similar features would be viewed as similar states and hence similar actions may be taken for both states. This helps us battle the aforementioned problems of traditional RL algorithms since

we can even learn about states that we have never visited in the past if the model has learnt to extract meaningful features from the grid during training. We use the following types of model architectures:

- FCNN
- CNN
- CNN with fixed kernels

The FCNN extracts features using a series of dense layers while a CNN extracts spatial features from the states, thereby exploiting the grid nature of the game.

5.1 Training methodology and its rationale

We train the networks using self play with the help of 2 networks, the online network, and the target network which we update after every 10 episodes. The update rule for the online network is as follows:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \Big(r + \gamma \max_{a'} Q'(s', a'; \theta') - Q(s, a; \theta) \Big)^2 \tag{1}$$

Here, Q, θ correspond to the model and parameters of the online network and Q', θ' correspond to that of the target network.
The model learns via self-training with the help of experience replay using a replay buffer that over-samples win/lose conditions (5 times more often than others) since those data points are crucial for learning. The replay buffer is a deque of size 1000 which only samples form the last 1000 data points since more recent data would based on more refined moves and are hence, more important and relevant for training. This also avoids oversampling older data points which are attained by taking sub-optimal actions which are less relevant since the model is now smarter. The models are self trained to avoid a skill ceiling which could arise if one architecture is significantly superior to the other leading to sub optimal training since both models are unable to outplay and hence, push each other enough. At first, we allowed the model to make invalid moves and penalized whenever it did, so that it learns to play knowing that it cannot make moves which are invalid. But it turned out that training the model such that it sees the board and only makes valid moves yielded better results, so we went ahead with this approach. We gave a reward of 5 and -5 for winning and losing respectively, which is not too low, which could make the change in the network insignificant, and not too high, which could change the network weights drastically potentially destroying the information learnt so far.

5.2 DQN Architectures

We used several DQN architectures and experimented with size, depth, padding (no padding and valid padding) 3x3 kernels, 4x4 kernels etc. We noticed that

CNNs were performing better than FCNNs so we explored more CNN architectures. The models are compiled using mean squared error loss and the Adam optimizer.

5.2.1 FCNN_self

This model is a Fully connected neural network (FCNN) architecture for playing Connect4 using reinforcement learning. The model architecture is as follows:

- Flatten Layer.
- Fully Connected Layers:
  - Dense layer with 256 units and ReLU activation function.
  - Dense layer with 128 units and ReLU activation function.
  - Dense layer with 64 units and ReLU activation function.
- Output Layer:
  - Dense layer with 7 units (output for Q-values).

The model is compiled using mean squared error loss and the Adam optimizer.

5.2.2 CNN_valid (CNN1)

This model is a convolutional neural network (CNN) architecture for playing Connect4 using reinforcement learning. The model architecture is as follows:

- Convolutional Layer:
  - 128 filters with a 3x3 kernel size and 'valid' padding.
  - ReLU activation function.
- Convolutional Layer:
  - 64 filters with a 3x3 kernel size and 'valid' padding.
  - ReLU activation function.
- Flatten Layer.
- Fully Connected Layers:
  - Dense layer with 128 units and ReLU activation function.
- Output Layer:
  - Dense layer with 7 units (output for Q-values).

The model is compiled using mean squared error loss and the Adam optimizer.

5.2.3 CNN_not_custom_64_4_64_2_64_64 (CNN2)

This model is a convolutional neural network (CNN) architecture for playing Connect4 using reinforcement learning. The model architecture is as follows:

- Convolutional Layer:
  - 64 filters with a 4x4 kernel size and 'same' padding.
  - ReLU activation function.
- Convolutional Layer:
  - 64 filters with a 2x2 kernel size and 'same' padding.
  - ReLU activation function.
- Flatten Layer.
- Fully Connected Layers:
  - Dense layer with 64 units and ReLU activation function.
  - Dense layer with 64 units and ReLU activation function.
- Output Layer:
  - Dense layer with 7 units (output for Q-values).

The model is compiled using mean squared error loss and the Adam optimizer.

5.2.4 CNN_not_custom_128_4_64_64 (CNN3)

This model is a convolutional neural network (CNN) architecture for playing Connect4 using reinforcement learning. The model architecture is as follows:

- Convolutional Layer:
  - 128 filters with a 4x4 kernel size and 'same' padding.
  - ReLU activation function.
- Flatten Layer.
- Fully Connected Layers:
  - Dense layer with 64 units and ReLU activation function.
  - Dense layer with 64 units and ReLU activation function.
- Output Layer:
  - Dense layer with 7 units (output for Q-values).

The model is compiled using mean squared error loss and the Adam optimizer.

5.2.5 CNN_8_only (CNN4)

This model implements a convolutional neural network (CNN) architecture for playing Connect4 using reinforcement learning. The model architecture is as follows:

- Custom Convolutional Layers:
  - 6 custom convolutional layers, each with a 3x3 kernel size.
  - Non-trainable weights based on predefined filters for specific patterns in the Connect4 game board:
    - **Left Filter**: Detects patterns where a player has three discs aligned to the left.
    - **Right Filter**: Detects patterns where a player has three discs aligned to the right.
    - **Top Filter**: Detects patterns where a player has three discs aligned vertically at the top.
    - **Bottom Filter**: Detects patterns where a player has three discs aligned vertically at the bottom.
    - **Right Diagonal Filter**: Detects patterns where a player has three discs aligned diagonally from the top left to the bottom right.
    - **Left Diagonal Filter**: Detects patterns where a player has three discs aligned diagonally from the top right to the bottom left.
- Convolutional Layers:
  - 128 filters with a 3x3 kernel size and 'same' padding.
  - Rectified Linear Unit (ReLU) activation function.
- Batch Normalization Layer.
- Dropout Layer with a dropout rate of 0.4.
- Convolutional Layer:
  - 128 filters with a 3x3 kernel size and 'same' padding.
  - ReLU activation function.
- MaxPooling2D Layer with a pool size of (2, 2).
- Batch Normalization Layer.
- Dropout Layer with a dropout rate of 0.4.
- Convolutional Layer:
  - 128 filters with a 3x3 kernel size and 'same' padding.

  - ReLU activation function.

- MaxPooling2D Layer with a pool size of (2, 2).
- Batch Normalization Layer.
- Dropout Layer with a dropout rate of 0.4.
- GlobalAveragePooling2D Layer.
- Fully Connected Layers:
  - Dense layer with 128 units and ReLU activation function.
  - Dense layer with 64 units and ReLU activation function.
- Output Layer:
  - Dense layer with 7 units (output for Q-values).

6 Supporting Classes

6.1 Connect4

The `Connect4` class represents the Connect4 game environment. It includes the following methods:

- `__init__`: Initializes the game environment with the specified parameters.
- `reset`: Resets the game to its initial state.
- `is_valid_move`: Checks if a given move is valid.
- `make_move`: Makes a move in the game.
- `check_winner`: Checks if there is a winner in the current state of the game.
- `is_board_full`: Checks if the game board is full.
- `step`: Takes a step in the game based on the specified action.
- `render`: Renders the current state of the game.

The class also provides properties for the observation space and action space dimensions.

6.2 DQNAgent

The `DQNAgent` class implements a Deep Q-Network (DQN) agent for playing Connect4. It includes the following methods:

- `__init__`: Initializes the agent with the specified parameters.
- `update_target_network`: Updates the target network with the weights of the online network.
- `remember`: Stores experiences in the agent's replay memory.
- `act`: Selects an action based on the epsilon-greedy policy.
- `replay`: Performs experience replay and updates the online network.

The agent uses a convolutional neural network (CNN) model to approximate Q-values for actions in the Connect4 game.

7 Tree algorithms

7.1 Minimax and negamax

Minimax algorithm is a tree search algorithm that explores the game tree by considering all possible moves for both players, recursively evaluating each resulting board configuration. The algorithm alternates between maximizing and minimizing players, aiming to maximize the score for the maximizing player while minimizing potential losses against an optimal opponent. Alpha-beta pruning enhances efficiency by eliminating branches of the game tree that cannot affect the final decision. It maintains two values, alpha and beta, representing the best scores found for the maximizing and minimizing players, respectively. When a move leads to a worse outcome than the current best option for the opponent, the subtree is pruned, reducing the number of nodes evaluated. This pruning technique significantly speeds up the search process, allowing the algorithm to explore deeper into the game tree within a reasonable time frame. We use rewards as the evaluating function for giving value estimate for a state. An ideal minimax tree would give us the optimal policy when built fully, but since we cannot have trillions of nodes in a tree, we limit the depth of search to 3(we only look at 3 actions in advance) and use a heuristic evaluation function . Kaggle's negamax also uses the minimax algorithm, except it combines the roles of the maximizing and minimizing players, exploiting the zero-sum property of the game. We built a custom minimax algorithm and found that it performs similar to our benchmark negamax.

7.2 Monte Carlo Tree Search (MCTS)

MCTS is an iterative search algorithm that progressively builds a search tree by exploring promising branches of the game tree. For each state seen during a game, MCTS builds a tree to give the best possible move. The algorithm is summarized as follows:

1. **Selection**: Start from the root node (current game state) and recursively select child nodes until a leaf node (unexplored or terminal state) is reached. Selection is typically done using an exploration-exploitation strategy called Upper Confidence Bounds (UCB). The UCB score is given by :

$$UCB(a) = \overline{X}_a + c\sqrt{\frac{\ln N}{N_a}} \tag{2}$$

   Where:

   - $\overline{X}_a$: Average reward of action a.
   - c: Exploration parameter (usually a constant like 2).
   - $\ln N$: Natural logarithm of the total number of time steps.
   - N_a: Number of times action a has been selected so far.

2. **Expansion**: If the selected leaf node has unexplored child nodes, expand it by adding one or more child nodes corresponding to possible actions from that state.

3. **Simulation (Rollout)**: Perform a simulation (also known as a rollout) from the expanded node by making random moves until a terminal state (win, loss, or draw) is reached. The outcome of the simulation is used to evaluate the quality of the expanded node.

4. **Backpropagation**: Update the statistics of all nodes visited during the selection phase based on the outcome of the simulation. This includes updating the visit count and the total reward accumulated for each node.

5. **Action Selection**: After a fixed number of iterations or computational budget, select the best action to play based on the child node with maximum average reward.

We run the algorithm for various iterations and compare the results.

7.3 AlphaZero

The alphazero algorithm combines the power of neural network to evaluate states and MCTS, designed to learn the optimal strategy for playing Connect4 by self-play and deep neural network training. The algorithm is described below: Initialization: Initialize the neural network model with random weights.

- **Self-Play**: Generate training data by playing many games of Connect4 against itself. During self-play, the algorithm uses Monte Carlo Tree Search (MCTS) to explore possible moves and select the best move according to the current neural network policy. This MCTS does not use

rollout to evaluate the win ratio, instead it uses the value estimate given by the neural network. To navigate down the tree, instead of UCB score it uses a PUCT score shown below:
$pUCT(s,a) = \frac{Q(s,a)}{N(s,a)} + c \cdot P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$

  - s represents the game state and a represents the action
  - $Q(s,a)$ is the action-value (Q-value) estimation for taking action a from state s
  - $N(s,a)$ is the visit count
  - $P(s,a)$ is the prior probability of selecting action a from state s
  - c is a constant determining the level of exploration.

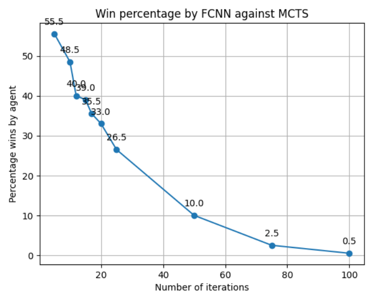
  During backpropogation, $Q(s,a)$ gets updated as follows: $Q(s,a) = \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$ where v is the backpropogated value. Each data point now contains the state as the input and reward of the self play- game along with updated P(s,a) value ($= \frac{N(s,a)}{\sum_a N(s,a)}$).

- **Neural Network Training**: Use the generated self-play data to train the neural network model. The neural network learns to predict the probability distribution of possible moves and gives a value estimate for each state that indicates the expected outcome of the game from a given board position.
- **Iteration**: Repeat the self-play and neural network training process iteratively, gradually improving the neural network's policy and value estimation capabilities.
- **Prediction** : The neural network is trained offline and during online prediction, given a state, our MCTS tells us which action to choose based on maximum score, which in turn is evaluated using the trained neural network.

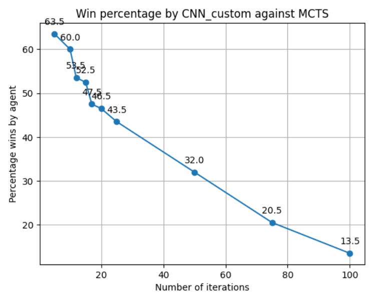
For each iteration of dataset generation, our MCTS plays 50,000 episodes with itself. This MCTS runs 100 iterations to give the value estimate. The neural network has 3CNN layers(64 filters), 2 fully connected layers(1280 and 256 units) and some batch normalization and dropout layers. The algorithm was trained for 60 epochs.

8 Results

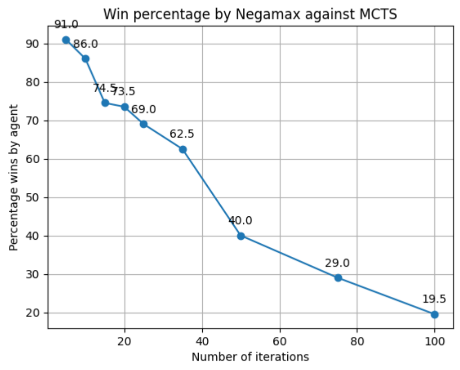
The performance and evaluation of performance of traditional Reinforcement Learning models was discussed in section 4. They were only able to visit less than 0.00001 percentage of the entire state space during training and hence were performing random moves during testing , hence they learnt very less despite being trained for many episodes. The Kaggle agent Negamax was set as the benchmark for our models. DQNs performed better than traditional RL methods. Among CNNs and FCNNs, the CNNs were performing better possibly because of the grid nature of the game which makes capturing spatial information as is done in CNNs very useful. Among the CNNs, we used 3x3 kernels without padding and 4x4 kernels with padding. The 3x3 kernels without padding performed better. The best among the models was the model that used some fixed 3x3 kernels which would observe immediate win/lose conditions. This could be because padding, which is done to allow for large kernels and deeper networks causes the grid to become an 8x9 grid with zeros in the edges possibly causing the model to assume that there are empty spaces in the grid where there are not leading the model to take misinformed decisions. The performance of our minimax was similar to that of Negamax. Since MCTS is an online algorithm, we tried restricting the tree search iterations and compared the performances. The graphs below show the winning percentage of some of our models plotted against many different iterations of MCTS. The CNNs are able to beat MCTS(10) but not MCTS(25). Negamax beats MCTS(25) but loses to MCTS(50). Alphazero, which in itself uses a variation of MCTS(100) is able to beat MCTS(100) and is declared the winning model.



(a) FCNN



(b) CNN



(c) Negamax

Figure 2: Performance of models vs MCTS iterations

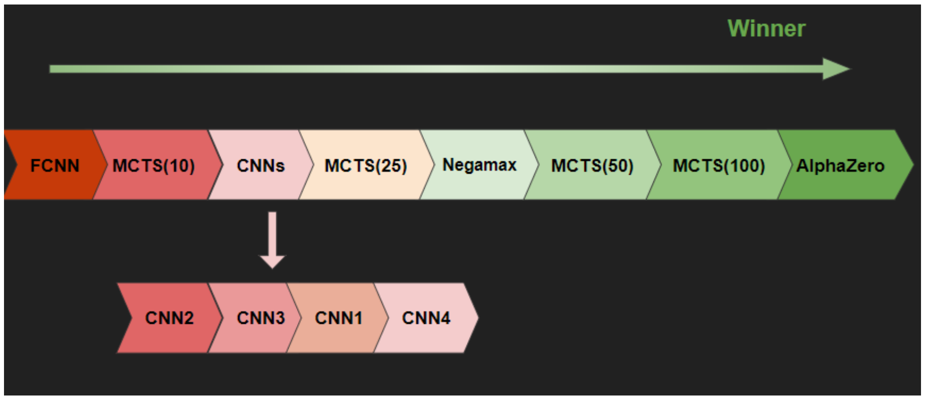


Figure 3: Summary of results

9 Conclusion

We tried training different models to play the game of Connect4 such as traditional RL methods, DQNs, tree based algorithms such as MCTS and AlphaZero. Negamax, which is another tree based learning algorithm is an in-built model of the Kaggle environment, which we used as a benchmark.
We observed that AlphaZero, a model that combines tree based learning like MCTS with deep learning, comes out as the best model of all that we considered. We also notice that the tree based learning algorithms like AlphaZero, MCTS and Negamax fare better than the DQNs (both FCNN and the CNNs). Traditional RL methods struggled to learn well due to the large state space and hence, we had to look for better alternatives like the DQNs and the tree based learning models.

10 Future developments

In terms of model improvements, possible future develops may include:

- Implementing a modified version of MCTS which expands every node visited in every simulation while building the tree from a given state. This may be feasible since the states are not very large and the episodes are not very long. This would drastically increase the space complexity of the algorithm, but could potentially yield better results since it utilizes all the information gained in the simulation step.

- Handcrafting and incorporating more fixed 4x4 kernels that could help detect win/lose conditions more easily and hence, make the model learn quicker.

- Building Deep Q-Networks which are symmetric by design, that is, if the input state is flipped horizontally, the output Q-value estimates should also be the same vector flipped horizontally, exploiting the symmetry of the game. To understand this, consider a state where the optimal action is to put a token in the rightmost column. Now if this grid was flipped left-right or horizontally, the optimal action would now be to put a token in the leftmost column.

11 References

- Reinforcement Learning for Connect4
- Monte Carlo Tree Search for Connect4
- Multi agent Reinforcement Learning in two-player zero-sum games
- The game is solved, white wins
- AlphaZero for Chess
- Minimax algorithm in Artificial Intelligence