Reinforcement Learning for Connect-4

Prateek Rath IMT2022017
Rudra Pathak IMT2022081
Soham Pawar IMT2022127

May 11, 2025

Abstract

This report presents a reinforcement learning approach to solving the game of Connect-4. The agent is trained using a Deep Q-Network (DQN) to learn optimal strategies for gameplay. Key concepts, implementation details, results, challeges and analysis are discussed.

1 Introduction

Connect-4 is a two-player strategy game where the objective is to connect four of one's own discs in a grid before the opponent. This project explores how reinforcement learning can be applied to develop an agent that learns to play Connect-4 effectively.

2 MDP Structure and the environment

Connect4 naturally gives us a MDP state, action, next-state, reward formulation. The state is simply the current board configuration i.e. a 6 x 7 grid. The actions are putting a circle or x (similar to that in tic-tac-toe) in a particular column. The next state is the new configuration obtained after the action is performed.
If we look at things from the perspective of an agent, the action of the other player can be understood as some stochasticity in the environment that leads to the next state. Thus after action a is performed in state s, the next state s' is obtained via some unknown probability transition matrix. Thus we are

in the model free setting.
The design of the reward function can vary. Typically, configurations that are favorable for the agent yield positive rewards while other configurations yield negative rewards. It's very difficult to tell whether a configuration is favorable or not and heuristics used often tend to have fatal flaws. We'll look at this in the challenges section.
For our purposes we use a sparse reward system. This also forces us to set gamma, the discount factor to be very close to 1(0.999).

- +x for a win
- +c for a draw
- -x for a loss

Here, x c and -x could be altered. We can also add a small punishment say -0.05 for taking too long to win. (Or a reward of +0.05 for lasting long)

3 More Details about the environment

The environment provides the agent with the following methods:

- get_board() - returns a copy of the current board state
- render() - displays the board in the terminal
- reset() - resets the environment
- get_available_actions() - returns the valid actions in the current board configuration
- check_game_done(player) - checks if player has won the game
- make_move(a, player) - performs action a on behalf of player a and returns next_state, reward

Other than this, the environment also keeps track of whose turn it is (player1 or player2).

4 Implementation

4.1 Network Architecture

We use several convolutional layers followed by some fully connected layers. This architecture is well-suited for Connect4 because:

- The convolutional layers allow the model to learn spatial patterns, which is essential for understanding the board layout and making strategic decisions.
- The deep architecture (several convolutional and fully connected layers) allows the model to learn complex relationships between the current state and possible actions.
- The activation function we use is leaky relu which is a generally recommended activation function.

4.2 Training the player: Auxillary functions

The batch size is typically between 128 and 512. The discount factor γ should be close to 1 as we want to maximize rewards in the long run(we want to win the game) and we don't care about immediate rewards. In fact immediate rewards are anyways 0 as we have chosen a sparse reward setting.
Two keys functions are needed while training:

1. *select_action(state, valid_actions, steps_done, training=True)*
This function selects an action using the epsilon-greedy policy. So it will either choose a random action or will choose an action that maximizes the Q-value of that state. As steps_done increases, the probability of choosing a random action (exploration) decreases exponentially and we converge to the optimal policy that chooses actions that maximize Q-values at each state.

2. optimize_model()
It takes up a mini-batch from the replay buffer and uses that to update the parameters. Here we use a smooth l1 loss to calculate the loss between the state-action values and the expected state-action values in that batch. We then use this loss to update our parameters. We use the adam optimizer as it is commonly used in reinforcement learning tasks.

4.3 Main Training Loop

1. First we use the select_action function to select an action from the agent's perspective.

2. This gives us the next state and the reward. We push the (next state, action , reward) tuple into the replay memory

3. We then select an action on behalf of the opponent. This can be done in several ways. We have used two variants namely random and minimax(with a small depth). Other variants such as a heuristic player, a noob player etc can also be added.

4. This gives us the next state and the reward again. We similarly push another tuple into the replay memory.

5. After dumping these two tuples into memory, we call the optimize model function. As specified earlier, this function updates the model's parameters using the replay memory.

6. Finally, ever TARGET_UPDATE iterations, we make the target network's parameters equal to the policy network's parameters.

7. We repeat the above steps for multiple episodes to complete training.

4.4 MCTS

1. Each Node tracks a move, visit count N, total reward Q, children nodes, and the game outcome. Node value uses UCT with exploration constant $\sqrt{2}$.

2. The tree policy (select_node) navigates the tree using the UCT value until a leaf is found. If expandable, new child nodes are added based on available moves.

3. A random playout is executed from the selected node using random valid moves until the game ends, then the result is evaluated.

4. Results are propagated up the tree, alternating reward values between players. Draws contribute a reward of 0.

5. The MCTS class wraps the entire flow—search() runs simulations within a time limit, best_move() picks the most visited child, and move() advances the tree after an opponent's move.

5 Experiments and Results

We decided to follow a curriculum learning approach wherein the model would progressively train against a series of opponents to get better. The initial idea was to have a minimax opponent with depth 1 to begin with and then increase the depth all the way. However, we quickly realized that this approach is infeasible as minimax with depths like 5 or 6 will take a very long time to compute. Hence we decided to start with a random player and a minimax player of depth 1 and then a minimax player of depth2. The players stronger than these could be rule based players.
In curriculum learning the agents trains against each opponent one by one till it can beat that opponent and we expect it to improve progressively.
After every few training steps we make the agent play 100 games and compute a win rate to see progress.

5.1 Random Opponent

First we made two random opponents play against each other for a large number of iterations(10000) and found the win rate to be 0.5533. This serves as our baseline. If reinforcement learning algorithms are able to achieve scores higher than this, then there has been some successful learning.
After training our agent for a large number of episodes(20000), we arrive at a player who plays very greedily and tries to stack coins vertically on top of each other to secure a quick win versus a random player. Due to the small penalty of -0.05 per move, it tries to win the game as soon as possible. Here is an image that illustrates one such victory.

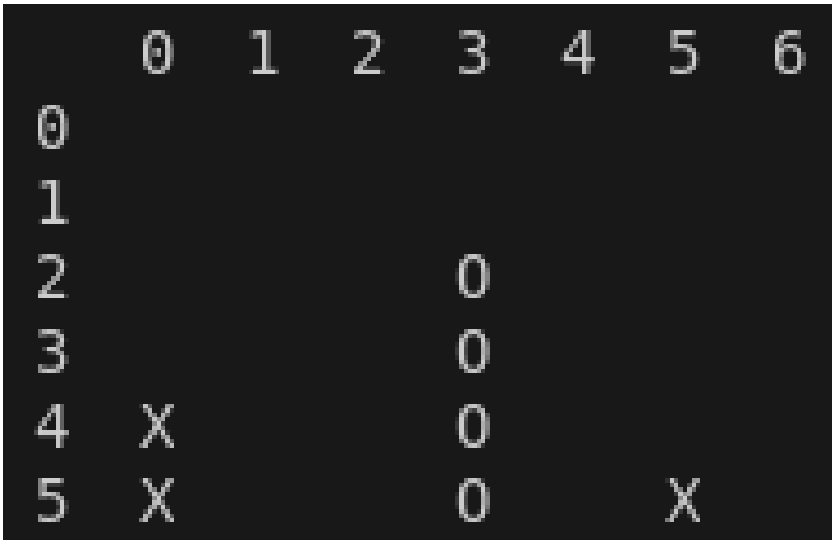

Figure 1: Quick Win

We also present a graph showing how the learning rate has improved over time.

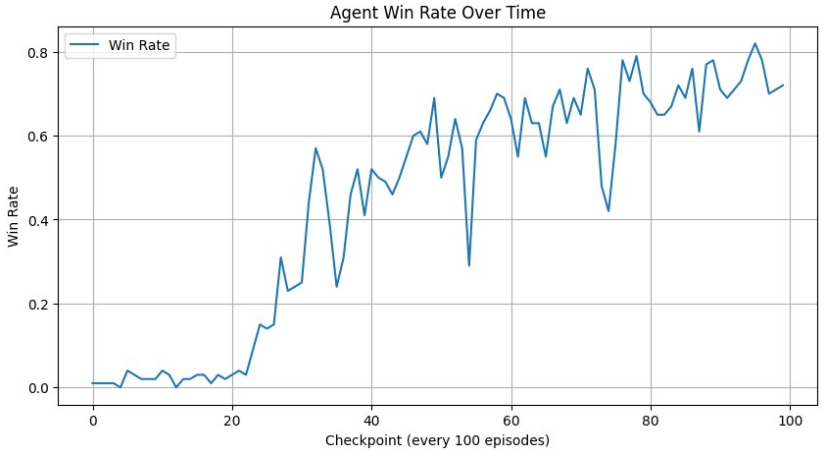

Figure 2: Win Rate vs Random Opponent while training

However, this same agent when tested against the win_block player fails miserably as the win_block player knows how to block immediate wins.

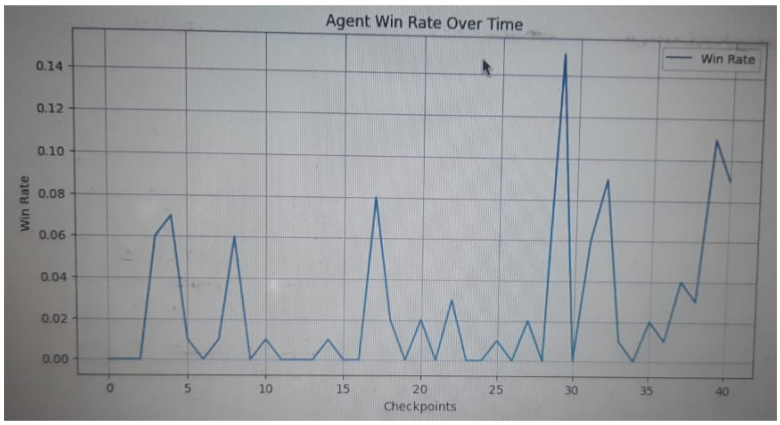

Figure 3: Erratic learning against win_block Player

5.2 Win Block Player

A win block player operates as follows. If it sees an immediate win for itself, it does the move right away. If it sees an immediate win for the opponent, it blocks it right away. Otherwise, it plays random moves. We see that this player actually is challenging to beat and the greedy strategy we learned previously will not suffice.
Here we tried different kinds of hyperparameter tuning to allow the DQN to converge and learn to beat such a player. Out of 100 games, around 5-12 games were victories that showed potential, involved double tricks or diagonal wins which a win block player cannot prevent except with a low probability. However, most of the learning curves looked like the figure below.

We were able to finally achieve a win_rate of around 0.3 against the wb player. We also found interesting double tricks, diagonal waiting wins and patterns like this in the wins experienced against the wb player. Different hyperparameter tuning lead to more fluctuations in win rate. The next section details some hyperparameter choices.

We also attempted to change the reward function to one that could provide immediate rewards.

- 'win': 100
- 'draw': 1
- 'lose': -500
- 'valid_2': 20
- 'valid_3': 30

Valid_3 and valid_2 refer to consecutive threes and twos formed while playing. This didn't work out. Again the graph was oscillatory as in the figure below.

Figure 4: Erratic win rate against win_block player with the new reward

Considering the data and results from the DQN network and the different reward functions tested, we next decided to change the DQN model. The original DQN model consists of multiple convolution layers and then a FFN to give the final actions. We suspect that this erratic behavior could also be due to the high complexity of the model. Therefore, we build a new DQN-Light model by removing a few convolution layers. As can be clearly seen from the graph 5, although the win-rates are erratic, there is an overall increase in the win-rate of the agent.

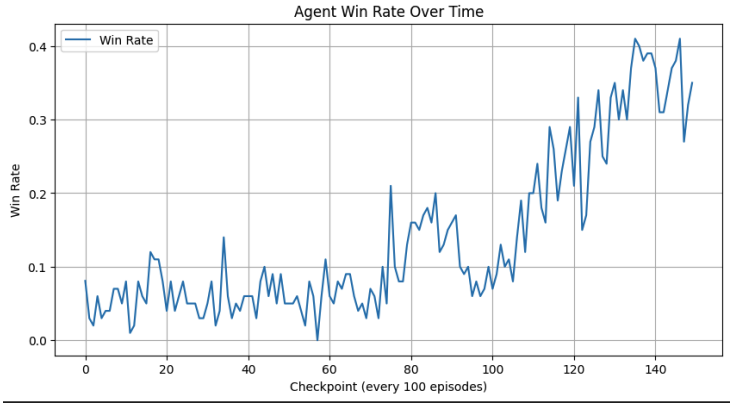

Figure 5: Erratic win rate against win_block player with the new reward and lighter DQN network with overall increasing win-rate

5.3 Minimax d1 Player

A minimax with depth 1 is weaker than a Win Block Player. Rather than doing random moves, it evaluates positions and does moves based on certain heuristics such as connecting to previous cells. Our agent(trained using win block) achieves a stellar score of 0.82 against a minimax player of depth 1.

5.4 Heuristic Player

A heuristic player is a very strong player. It is primed to not only look at least two moves ahead but also think in terms of double trick lookahead. It thinks almost exactly as I would think while playing a connect4 game. If it isn't forced to block a double trick or an immediate win for the opponent, it tries to create it's own double tricks or forms the most 'open' threes or twos. Open threes or twos refer to 3 or 2 cells of the same color in sequence that pose a threat to the opponent. For example '1 1 1 0' is an open three, but '2 1 1 1 2' is a completely closed 3. Without going into messy details, the heuristic player tries to maximize the number of open threes and twos in a position where there are no double tricks or immediate wins(for it or its opponent).
However, training an agent to play against a heuristic player is extremely hard. The main problem is that exploration doesn't allow the agent to see a single win in over 5000 iterations.

5.5 MCTS

It was difficult to quantify the results for this algorithm as this is a very time taking process. As search times increase, MCTS performs increasingly well and quickly beats the win block player. However, the search times need to be around 2000 seconds to achieve 100000 rollouts. This clearly isn't very feasible. // In general, MCTS easily beats the random player with a search time of about 8 seconds per move. Again, it appears to be following the greedy strategy of stacking vertically.

6 Challenges

1. The main challenge was hyperparameter tuning. It was difficult to know when the network was actually learning. The targets could be moving too fast or too slow. This could be due to an incorrect TARGET_UPDATE value or due to a small learning rate and it is very difficult to discern. Looking at the Q-values manually did help at times. We found that the Q-values were extremely close for all actions despite long training periods.

2. Another challenge was the limited methods available. Actor-Critic methods were out of the picture as there is no regular critic update possible after every timestep. We are bound to use a sparse reward system to prevent reward-hacking. Moreover the sample inefficiency of policy gradients would take forever to train, making it difficult to analyse results and progress.

3. DQNs don't guarantee convergence and suffer from the deadly triad problem. We could only hope for the method to work via exploration and exploitation. Despite enough hyper-parameter tuning and increasing exploration, we weren't able to see strict improvements in win_rates.

Here is a table that shows some attempts to tune hyperparameters and their results.

Opponent	Episodes	Batch Size	LR	Target Update	Eps Decay	Score
Random	20000	256	1.00×10^{-3}	10	3000	0.84
WB	20010	256	1.00×10^{-3}	10	3500	fluctuate
WB	40010	256	1.00×10^{-3}	10	6500	0
WB	21050	512	1.00×10^{-3}	100	5000	0.18
WB	10010	512	1.00×10^{-3}	200	3500	0.0–0.12
WB	30050	512	1.00×10^{-3}	100	5000	0.3
WB	15040	512	1.00×10^{-4}	1000	3500	0.0

Table 1: DQN training results against different opponents and hyperparameters

Small update rates like 1e-4 don't work well as the win_rate remains zero even after several iterations of training. It appears that coupled with the clipped gradients, a small learning rate doesn't help at all. Even if the target_update value is drastically increased, no learning is shown and the win_rate remains 0.

This is seen in the figure below.

```
starting training
------------------------------------------------
target update 1000
learning rate 0.0001
num episodes 15040
epsilon decay 3500
batch size 512
------------------------------------------------
episode  0
recent rate is  0.0
<ipython-input-12-301128821b08>:73: UserWarning:
  state_batch = torch.tensor(state_batch, dtype=
episode  500
recent rate is  0.0
episode  1000
recent rate is  0.0
episode  1500
recent rate is  0.0
episode  2000
recent rate is  0.0
```

Figure 6: Small learning rate consequences

7 Conclusion and Further Possibilities

We achieved commendable results against a random player. The agent successfully learnt a greedy strategy that beats the random player by continuously playing on the same column to secure a win.
Against a smarter Win Block player, the win rate is as good, but we can still see that there is some learning. Double tricks are a sequence of one or more threes that when formed force a win from that position. Diagonal wins cannot be seen directly with one look-ahead. These are the new kinds of winning strategies we expect the agent to learn after training for a while. For instance figure 4 shows a double trick that the player completes after repeatedly playing on column 5.
The loss penalty of -0.05 might just be preventing our model from lasting longer. It might give up quickly instead of trying to fight for a win. Maybe we should encourage it every time it blocks a win. Again this could lead to another form of reward hacking where we try to constantly block the opponent instead of trying to win ourselves.

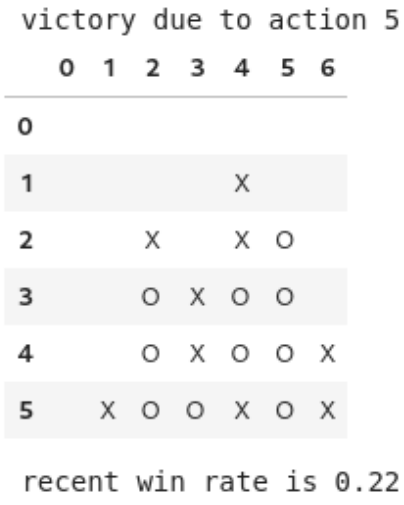

Figure 7: Double Trick

Further Improvements might be possible. We can experiment with more hyperparameters, train for more iterations etc. However, it must be noted that solving the problem using only a DQN algorithm may not be possible. The deadly triad problem is often present in such scenarios due to the trio

of bootstrapping, off policy and function approximation.

8 References

- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*.
- https://docs.pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
- https://github.com/neoyung/connect-4