Department of Computer Science
Ashoka University

Reinforcement Learning: CS-5410

Midterm Project Report

Collaborators: None **Name:** Jyotirmay Zamre

---

1 Introduction (Link)

For my midterm project, I decided to apply reinforcement learning to train an agent to play the game of Connect-4. Connect-4 is a 2-player game similar to Tic-Tac-Toe with the following differences:

1. The official board has 7 columns and 6 rows
2. A player requires 4 tokens, either horizontally, vertically, or diagonally, in a row in order to win
3. Gravity exists as a concept in this game so that a player places a token in a column and the token falls to the lowest empty space in that column

Like Tic-Tac-Toe, this is a zero-sum game. This means that if one player wins, the other automatically loses. Therefore, it is possible to play in a way solely to prevent the other player from winning.

1.1 Formulation as an MDP

The game can be easily defined as a Markov decision process (finite horizon since the game ends) as follows:

1. **State Space**: The state space, as one can imagine, is extremely large for Connect-4. Each space in the board can either be empty, have player 1's token, or player 2's token. If we consider invalid states as well (states where a player has already won), we can get a combinatorial upper bound of 3^{42} states. The total number of possible states is 4531985219092 (Link).
2. **Action Space**: A token can be placed in any of the 7 columns. It can be defined as $A = \{0, 1, 2, 3, 4, 5, 6\}$
3. **Rewards**: The only rewards we can conceive of are terminal rewards. If player 1 wins a game, it gets a reward of 1. If it loses, it gets a reward of -1. If the game ends in a draw, the reward can be decided to be anything in the interval $[-1, 1]$
4. **Transition Probability**: This is a 2-player game so the transition probability distribution will depend on what actions are taken by player 2. This makes it impossible to determine beforehand.

1.2 Reinforcement Learning

The goal of applying reinforcement learning is to learn a deterministic policy that an agent can apply. We ideally want to optimize the value function such that:

$$V^*(s) = max_a(r(s) + \gamma \sum p(s'|s,a)V^*(s'))\forall s$$

From this, we can extract the optimal policy that can be applied to Connect-4. However, there is one glaring issue. We do not have access to a transition function which provides the probabilities of moving to different states. Therefore, we cannot directly throw basic reinforcement learning techniques such as value iteration or policy iteration at the problem. We need to make use of reinforcement algorithms that utilize episode sequences of the form - $s_0, a_0, s_1, a_1, s_2, a_2, ...$ where s_{n+1} is reached after taking action a_n in state s_n. These algorithms rely on experience (such as Monte Carlo methods or Temporal Difference Learning)

2 Methodology

I decided to utilize Q-Learning to tackle this approach. Q-Learning is an off-policy algorithm in the suite of Temporal Difference learning algorithms. While Monte-Carlo methods wait until the entire episode sequence has ended to generate a distribution and update the value function, Q-Learning only waits one step to update the value function. The update step is given as:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R + \gamma \cdot max_a Q(s',a') - Q(s,a)]$$

where $Q(s,a)$ is the Q-value for state s and action a, α is the learning rate, R is the immediate reward received for taking action a in state s, γ is the discount factor, and $Q(s',a')$ is the best possible reward achievable from the next state s'. Therefore, we need to keep track of the following things:

1. We need to store a Q-table which will store the values for each state and action pair. This is not feasible in our case since the state space itself is so large.

2. We need to be able to generate high quality episodes that will be used to train the agent

In order to tackle the first point, since we cannot store the Q-table, we need some kind of approximator that can be used to estimate the table. We cannot use simple function approximators because the Q table for Connect 4 is not smooth. Therefore, we can look at machine learning approaches such as neural networks to do so. This brings us to the concept of Deep Q Learning.

2.1 Deep Q Learning (Reference 1, Reference 2)

Deep Q Learning uses neural networks to approximate the value function where it would be impossible to store an actual table. A trained neural network would allow the agent to estimate the values at each state for all given actions, allowing it to determine the best action it should take. However, we need to find a way to actually train the neural network.

2.2 Training

We need to utilize some sort of loss function. This loss function will tell us how far off the network's predicted value is compared to some target value. The actual Q-value can be calculated by using the network's prediction along with the max discounted return from the next state. We do the following steps:

1. We utilize two networks. One is the main network that will be trained and another is a target network.

2. The main network will be used to make original guesses and the target network will be used to make better guesses compared to the main network. This will be the basis for the loss function used to train the main network

3. Every few episodes, the weights of the target network will be updated to equal those of the main network.

2.3 Epsilon Greedy Strategy

Initially, the agent (and the network) knows nothing about the environment. An issue that it can run into is finding a strategy that gives it the maximum return and sticking with it for the rest of the network's training. This prevents the agent from experiencing other states. Given that the state space is so vast, there are bound to be many states that the agent will never encounter. In order to resolve this, we use something called an *epsilon-greedy strategy*. This strategy does the following:

1. With probability ϵ, the agent chooses a random action

2. With probability $1 - \epsilon$, the agent chooses the action that gives it the highest return

3. Over time, the value of ϵ decays so that as the agent goes through its training phase, the probability of choosing a random action reduces and the agent can rely more on what it has learnt.

In my implementation, I have used an exponential decay to model the epsilon-greedy strategy.

2.4 Experience Replay

In order to reduce correlation during training, experience replay is a method that is utilized. Each transition that is observed by the agent is stored. The network is then trained on a certain number of random samples from this list. This prevents the network from training on consecutive transitions, allowing it to be more robust in the way it learns.

3 Main Algorithm and Decisions

The main decisions and parameters are as follows:

1. Since this is a two player game, we can generate episodes only by making our agent play against another player. For this purpose, I have utilized a player that makes random decisions to play the game. This is not ideal and the opponent should have been another trained agent (such as using Monte-Carlo Tree Search or Minimax).

2. I used a convolutional neural network with the following architecture: 2 convolutional layers with a 5x5 kernel which are then followed by 3 fully connected layers. Each layer except for the last output layer is activated using *ReLu*. The optimizer used to train the network is *Adam* with a learning rate of 0.001. Mean Squared Error is used as the lass function

3. The target network is updated every 15 episodes to have the weights of the main network

3.1 Training Algorithm

The algorithm used for training is as follows:

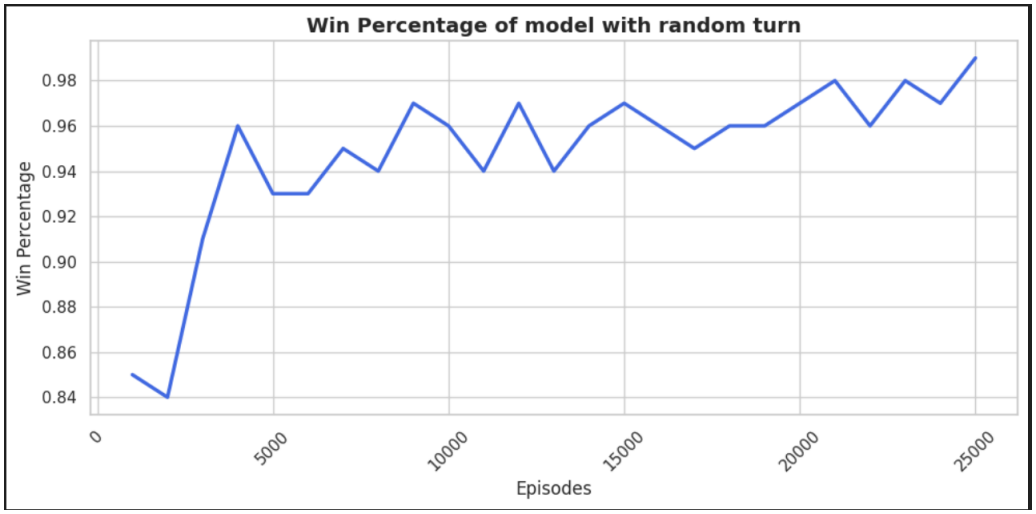
```
Loop through number of episodes
    Reset environment
    Randomly select which player makes the first move
    while episode not finished:
        Use epsilon-greedy strategy to pick an action
        Obtain new state, increment steps done (for decay)
        Check if the game is finished and add experience to memory
        Make the random agent take an action
        Check if the game is done and add experience to memory
        Add a non-terminal experience with a negative reward multiplied by moves played
        Optimize main network
    if N episodes have passed, update target network's weights
```

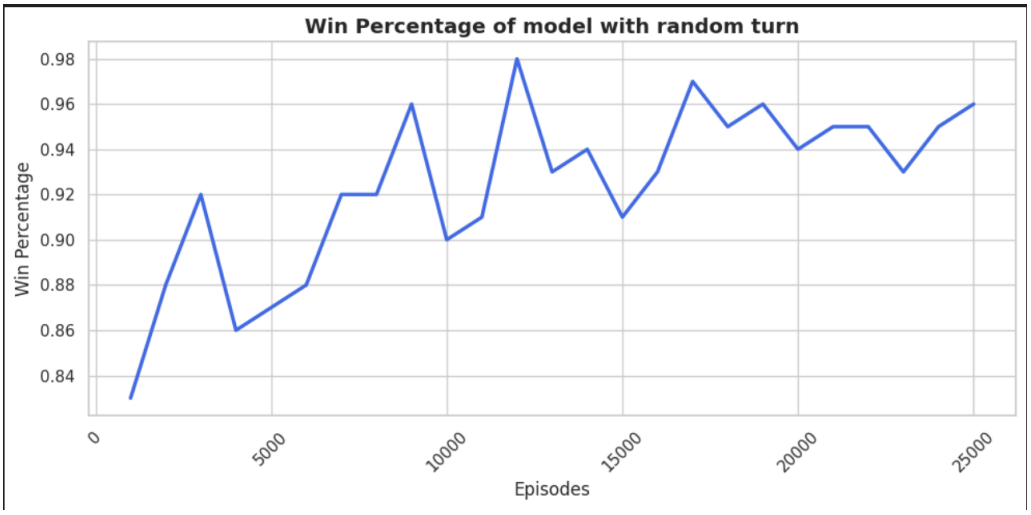
3.2 Novel Contribution

A lot of implementations of Deep Q Learning let the agent always make the first move in each episode. This is not the best strategy as the agent should experience episodes where it is going second. This is because there are different ways to guarantee wins depending on whether you go first or second. Therefore, for each episode, I am randomly choosing whether my agent or the random opponent goes first in the game.

4 Findings

I need to compare two different models: one where the agent always goes first during training and one where the player that goes first is randomized. In order to compare, I decided to calculate the winning percentage of the agent against the random agent every 1000 episodes. Every 1000 episodes, they will play a total of 100 games, and the winning percentage is then recorded. This is done for both models.





From these two graphs, it is clear that the model where the first turn was randomized converged to a high win percentage fairly early as compared to the model where my Agent went first every time. At the end of 25000 episodes, my model also converged to a higher win percentage and at a much smoother rate. This shows that providing such diversity in training made my proposed agent more robust and faster to learn. This has certain implications as we can train for fewer episodes for my model as compared to the other, saving compute time.

5 Conclusion

In conclusion, using my proposed strategy to train a deep Q network proved to be beneficial to the learning process. I would like to extend this by training against a more refined opponent and then observe results.