Connect 4 with Deep Q-Learning

CIS 5603 - Artificial Intelligence
Prof. Pei Wang

XiaoHui Kang
Donye Wakefield
Dec 15, 2025

Introduction

This Project implements a Deep Q-Network (DQN) agent to play the game Connect 4. The goal of the project was not only to train an agent that performs reasonably well, but also to understand the entire learning process, including environment design, reward shaping, training stability, evaluation, and practical debugging. Instead of using an existing library or environment online, we implemented the full system from scratch. This includes the Connect 4 environment, the replay buffer, the deep neural network, the training loop, the evaluation script, and finally a simple graphical user interface (GUI). By doing this, I was able to clearly see how different components of reinforcement learning (RL) interact with each other. Throughout the project, we encountered multiple issues such as unstable training, poor evaluation results, incorrect reward signals, and unexpected agent behavior. Solving these problems helped us understand RL beyond theory and formulas. This report describes the system design, the problems encountered, and the lesions learned during the learning process.

Environment Design

The first step of the project was to design a Connect 4 environment. The environment controls the board state, available actions, turn switching, reward calculation, and game termination. The board is a 6x7 matrix: 1 represents the agent's piece, -1 represents the opponent's piece, and 0 represents an empty cell. The agent always plays first. After each agent moves, a random opponent makes a move. This setup allows the agent to learn basic strategies while keeping the opponent simple. Instead of using the raw board matrix, we converted the board into a two-channel input. One channel shows where the agent's pieces are, and the other

channel shows where the opponent's pieces are. This format works better with convolutional neural networks (CNNs), which is what’s used as the backend of the DQN, because the network can learn patterns separately for each player. The function board_to_cnn_input() performs this conversion automatically before feeding the state to the neural network.The environment has several key methods. The reset() method clears the board and starts a new game. The step() method takes an action (which column to drop a piece into) and returns the next state, the reward, and whether the game is done. The available_actions() method returns which columns still have empty spaces. The check_winner() method checks all possible winning conditions: horizontal, vertical, and diagonal lines of four pieces. One important decision was the reward structure. We gave +1 reward for winning, 0 reward for a draw, and -2 reward for illegal moves. At first, we tried giving small negative rewards for every move to encourage the agent to win faster. However, this made training unstable because the agent would sometimes prefer losing quickly over winning slowly. We removed these step penalties and the training became more stable

Replay Buffer and Experience Replay

The replay buffer stores past experiences so the agent can learn from them multiple times. Each experience contains the state, action taken, reward received, next state, and whether the game ended. We store up to 50,000 experiences in a deque data structure. When the buffer is full, old experiences are automatically removed. During training, we randomly sample batches of 64 experiences from the buffer. This random sampling is important because it breaks the correlation between consecutive game states. If we trained on experiences in the order they happened, the agent would overfit to recent games and forget earlier lessons. Random sampling helps the agent learn more general strategies. At first, we had a bug where we started training before the buffer had enough experience. This caused errors because we tried to sample more experiences than existed. We fixed this by checking if the buffer size is at least 64 before starting any training step. This small check prevented many crashes early in development.

Deep Q-Network Agent Architecture

The primary learning component of this project is a Deep Q-Network (DQN) agent that would be designed to learn effective policies for playing Connect 4. It is important to explain how

the DQN framework works so that the model becomes clearer. A DQN combines Q-learning with deep neural networks, allowing the agent to approximate the action-value function Q(s,a) for high-dimensional state spaces that would otherwise be somewhat infeasible to represent using other more tabular methods. This Q(s,a) would be represented as Q-values, so effectively the DQN estimates the expected cumulative reward of placing a piece in any of the seven columns given the current configuration of the Connect 4 board. The neural network architecture used for function approximation is specifically tailored to exploit the spatial structure of the board. Each game state is represented as a tensor of shape (2,6,7), where the 2 channels correspond to the current player's pieces and the opponent's pieces. This encoding allows the model to distinguish between friendly and opposing tokens while also preserving the two-dimensional layout of the board. Such a representation is critical for learning spatial patterns, including vertical stacks, horizontal connections, diagonal threats, etc. The network begins with a convolutional component composed of three two-dimensional convolutional layers. The first convolutional layer applies 32 filters of size 3x3 with padding, followed by a ReLU activation function. This layer is intended to capture low-level spatial features such as adjacent pieces and simple alignments. The second convolutional layer increases the number of filters to 64, enabling the network to detect more complex patterns such as partial winning configurations and early-stage threats. A third convolutional layer with 64 filters further refines the extracted features, allowing the model to combine lower-level patterns into higher-level representations. Pooling layers were intentionally omitted from the architecture in order to preserve the full spatial resolution of the board, which basically means reducing dimensions through pooling would result in shrinking the board and therefore losing critical positional information in a relatively small game grid already. Following the convolutional layers, the output tensor is flattened and passed into a fully connected component of the network. The flattened feature vector has a size of 64×6×7, which is mapped to a hidden layer consisting of 256 neurons with ReLU activation. This dense layer enables the network to integrate spatial features across the entire board and should be able to reason about global board configurations. The final output layer contains seven neurons, one for each possible action corresponding to dropping a piece into one of the seven columns. Each output neuron represents a predicted Q-value for the associated action, allowing the agent to compare and select among legal moves. Lastly, in order to stabilize training, this DQN agent maintains two separate networks: a primary network and a target network. The primary network is updated frequently through gradient descent, while the target network is updated less often by copying the weights of the primary network. This separation should help mitigate the instability that can arise when the same network is used

both to select and evaluate actions, which is apparently a known issue in standard Q-learning with function approximation.

Training and Learning Strategy

Training the DQN agent would mean following an episodic RL framework, which means each episode corresponds to a full game of Connect 4. During training, the agent plays against a random opponent, which should provide a simple yet diverse baseline adversary. At the start of each episode, the environment is reset and the agent begins interacting with the game by selecting actions according to an epsilon-greedy policy. This strategy is supposed to balance exploration and exploitation by allowing the agent to choose random actions with probability (epsilon), while selecting the action with the highest predicted Q-value otherwise. The epsilon value is initially set to 1.0, resulting in fully random behavior at the beginning of training. This should encourage a more broader exploration of the state-action space and prevent the agent from prematurely converging to suboptimal strategies. Over time though, epsilon decays in a multiplicative manner toward a minimum value of 0.1, gradually shifting the agent's behavior toward exploitation of learned policies while still retaining some degree of randomness. This decay schedule allows the agent to refine its strategy based on accumulated experience while also avoiding the problem of overfitting too early. During each step of gameplay, the agent will store its experiences in the replay buffer. For each sampled batch from the replay buffer, the agent computes predicted Q-values for the actions taken using the primary network. Target Q-values are computed using the target network by selecting the maximum predicted value for the next state and applying the Bellman equation, which means the loss is calculated as the mean squared error between the predicted Q-values and the target values, and the network parameters are updated using the Adam optimizer. A discount factor (gamma) of 0.99 is used to prioritize long-term rewards, which is hopefully appropriate for a strategic game like Connect 4 where delayed outcomes such as setting up future wins are critical. After completing all training episodes, the learned model parameters are saved to storage, which would help allow the trained agent to be evaluated independently of the training process.

Model Evaluation and Performance Assessment

After training was completed, the performance of the Deep Q-Network agent was evaluated by having it play a series of games against a random opponent using a separate

evaluation script. During evaluation, the exploration was disabled by setting the agent's epsilon value to zero, ensuring that all actions were selected greedily based on the learned Q-values rather than random exploration. A total of 100 games were played, and the outcomes were recorded in terms of wins, losses, and draws. The evaluation results showed that the trained agent achieved 31 wins out of 100 games, corresponding to a win rate of 31%, while losing 69 games and producing no draws. These results indicate that, although the agent was able to learn some non-trivial strategies and even occasionally defeat the random opponent, its overall performance remained relatively weak. Given that a random agent would be expected to win approximately half of its games against another random player, the observed win rate suggests that the learned policy was not robust enough to consistently outperform the baseline behavior. Inspection of the final board states from several evaluation games provides additional insight into the agent's behavior. In many cases, the agent demonstrated an ability to form vertical or diagonal sequences early in the game, indicating that it had learned some basic spatial patterns relevant to Connect 4. However, the agent seemed to have frequently failed to recognize imminent threats from the opponent or missed opportunities to block winning moves. This suggests that while the convolutional neural network was able to extract these local board features, the agent struggled to apply enough reason effectively about the longer-term consequences and perform multi-step planning, which are critical in Connect 4.

Several factors likely contributed to the observed performance limitations. One major challenge was the simplicity of the opponent used during training and evaluation. While training against a random opponent allows the agent to learn fundamental mechanics of the game, it seems to not expose the agent to consistently strong strategies. As a result, the learned policy may have overfitted to suboptimal opponent behavior, and as a result, fail to generalize when faced with more strategically diverse situations. Additionally, the reward structure used during training was pretty sparse, with rewards primarily assigned at terminal states for wins or losses. Sparse rewards can slow down learning and make it difficult for the agent to correctly assign credit to intermediate actions that contribute to any long-term success. Another challenge encountered during training was the instability in the learning process. Although experience replay and a target network were implemented to improve stability, fluctuations in the loss values were still observed throughout training. This behavior is apparently common in deep RL and reflects the non-stationary nature of the learning target, as the agent's policy continuously evolves. Furthermore, the number of training episodes used was indeed limited due to computational constraints, even when training in environments such as Google Colab. A much

larger number of training episodes would likely be necessary for the agent to converge toward a stronger policy. Despite these limitations however, the evaluation results do demonstrate that the implemented DQN was functional and capable of learning from interaction with the environment, which shows progress. The agent showed measurable improvement over purely random behavior and successfully integrated convolutional feature extraction with Q-learning.

Future Improvements and Conclusion

There are several clear directions for future improvement like increasing the number of training episodes and adjusting the hyperparameters such as the learning rate, batch size, and epsilon decay schedule, which could lead to more stable learning and improved performance. Enhancing the reward function to include intermediate rewards for advantageous board positions or successful blocking moves may also help guide the agent toward better strategies. Additionally, training against stronger or adaptive opponents, such as maybe a heuristic-based agent or through self-play, could significantly improve the robustness and generalization ability of the learned policy. Also worth noting, there are more advanced extensions of DQN, such as the Double DQN or Dueling DQN architectures that could be explored that apparently would help even more in reducing overestimation bias and improve the learning efficiency. Overall though, while the current results indicate that the agent has not yet achieved not even close to strong competitive performance, the evaluation does highlight both the progress made and the challenges inherent in applying deep reinforcement learning to complex, strategic board games.