Deep Q-Learning for Complex Decision-Making: A Case Study with Connect Four

Thomas Favoli
thomas.favoli@student-cs.fr

Abstract

This project explores different approaches to solving games with large search spaces, such as Connect Four. Dynamic programming and Monte Carlo methods are effective for games with small search spaces, but may not perform well for larger games. Reinforcement learning, particularly deep Q-learning, is a more flexible and adaptable approach that can handle large search spaces and complex decision-making processes. The project also presents the results of training and simulating games between two deep Q-learning players using MLP and CNN. GitHub code

1. Introduction

Connect Four is a popular two-player game that is played on a vertical board with seven columns and six rows. Players take turns dropping colored discs into the columns from the top of the board, with the goal of being the first player to connect four of their colored discs in a row either horizontally, vertically, or diagonally.

Connect Four is a game that may seem simple, but actually offers significant strategic depth. By targeting specific slots on the board, players can increase their chances of winning. The central slots are particularly valuable, as they provide more opportunities to create a line of four. With many possible outcomes and numerous potential actions, players can employ various tactics to reach their goals. Our objective is to use reinforcement learning to identify the optimal strategies for succeeding in this (Connect Four) Markov Decision Process.

2. Problem description

In Connect Four, the action space is relatively small, as players have only seven possible moves at any given point, involving the placement of a game piece in one of the seven columns on the board. The number of moves available at a specific state depends on the number of full columns on the board, which we determine by calculating the difference between seven and the number of full columns.

There are three possible states that each individual grid square in the Connect Four game can be in: red, yellow, or empty. Considering that the game board consists of 7 columns and 6 rows, this leads to a rough estimate that there could potentially be 3^42 distinct combinations of grid states. However, this upper bound is very crude and does not take into account the game's specific rules and restrictions.

For games with a finite number of possible actions (including Connect Four), it is possible (theoretically) to create a game tree that represents every possible state of the game.

In this tree, each node corresponds to a possible game state, while the internal nodes at even depths represent either the initial state of the game (i.e., the root node) or a state resulting from a move made by the opponent. When a state is game ending (either because the board is full or because four tokens have been connected), it is considered a leaf node in the tree, and is not further expanded. Each leaf node is assigned a score based on the outcome of the game.

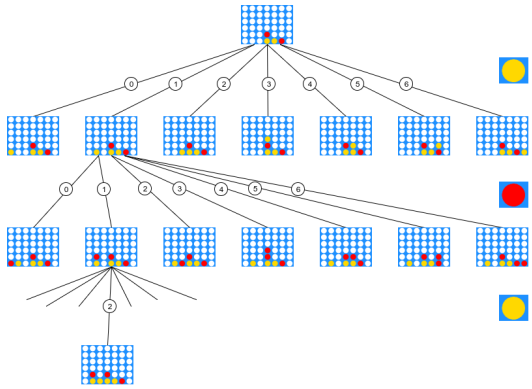

Figure 1 : An example of a path to a game-ending state in a game subtree [1].

By assigning values to each leaf node and propagating these values up the tree, the best possible move can be determined. However, constructing a game tree can be time-consuming and may not always be feasible due to computational limitations. As an alternative, a minimax algorithm can be used to minimize the maximum potential losses [1] for the player by assuming that their opponent will always make the best possible move, but we still face computational limitations.

This approach is very effective for games with relatively small search spaces, but become impractical for games with large search spaces, such as Connect Four. In that end, reinforcement learning is a more flexible and adaptable approach that can be used to solve games with large search spaces and complex decision-making processes.

Reinforcement learning is a machine learning technique that enables an agent to learn through trial and error. In the case of a game like Connect Four, the agent would play against itself or against human players, and use the outcomes of those games to update its strategy over time.

To apply RL to Connect Four, we can model the game as a Markov decision process (MDP), where the state of the game is the current configuration of the board, the actions are the possible column choices for the player to drop their disc, and the reward is +1 for a win, 0 for a tie, and -1 for a loss.

3. Dynamic Programming and Monte-Carlo Methods

Dynamic programming (DP) is a method that can be used to solve MDPs by recursively computing the optimal value of each state and the optimal policy that leads to that value [2]. However, DP becomes impractical for large MDPs like Connect Four because it requires storing and updating the value of every possible state, which is exponential in the size of the state space.

Monte Carlo methods, on the other hand, do not require a complete model of the MDP and can be used for environments where the dynamics are unknown or the state space is too large to be represented [2]. Monte Carlo methods generate rollouts of the agent's interactions with the environment and use these to estimate the value of states and the optimal policy.

In the case of Connect Four, we can use Monte Carlo tree search (MCTS), which is a variant of Monte Carlo that builds a search tree by selecting actions based on their estimated value and exploring promising branches of the tree.

MCTS works by simulating a sequence of actions starting from the current state of the game and selecting each action according to a policy that balances exploration and exploitation. The result of each simulation is a sequence of states, actions, and rewards, which can be used to update the estimated value of each state in the search tree.

By repeatedly simulating episodes of the game and updating the search tree, MCTS can converge to a good policy that maximizes the expected reward. In fact, MCTS has been used to achieve superhuman performance in several board games, including Go and chess. Therefore, Monte Carlo methods can be used to learn a good policy (using MTCS) for Connect Four, but they may require a large number of simulations to converge to a good policy.

We could have chosen to implement this method in our project but as demonstrated later why, we would rather implement a Deep-Q-learning method.

While DP and Monte Carlo methods can be useful for solving certain types of problems, they have limitations when it comes to large-scale problems like Connect Four. To address these limitations, researchers have developed more sophisticated reinforcement learning algorithms that combine elements of both DP and Monte Carlo methods.

For instance, temporal difference (TD) learning is an RL algorithm that uses an estimate of the value of the next state to update the estimate of the value of the current state, which can help to speed up convergence [2]. Temporal-Difference (TD) Learning shares similarities with Monte-Carlo methods in that it is model-free and learns from experience in episodes. TD learning has the additional advantage of being able to learn from incomplete episodes, which eliminates the need to track the episode until termination.

4. Temporal-Difference Learning

SARSA stands for State-Action-Reward-State-Action, and it is a type of reinforcement learning algorithm that learns a policy for a given environment [2].

The SARSA algorithm is an on-policy method, which means that it learns the value of the current policy while following that policy. The algorithm is based on the idea of Q-learning, which is to learn the optimal action-value function for a given policy.

The SARSA algorithm works by following an epsilon-greedy policy, where the agent chooses the best action with probability (1-epsilon), and a random action with probability epsilon. The algorithm updates the Q-values after each step, using the following update rule:

Q(s,a) <- Q(s,a) + alpha * (r + gamma * Q(s',a') - Q(s,a))

where s is the current state, a is the current action, r is the reward received after taking the action, s' is the next state, a' is the next action, alpha is the learning rate, and gamma is the discount factor.

Q-learning, on the other hand, is an off-policy method, which means that it learns the value of the optimal policy while following a different policy [2]. The Q-learning algorithm is based on the idea of Q-values, which represent the expected value of taking a certain action in a certain state.

The algorithm works by updating the Q-values using the following update rule:

Q(s,a) <- Q(s,a) + alpha * (r + gamma * max_a'(Q(s',a')) - Q(s,a))

where s is the current state, a is the current action, r is the reward received after taking the action, s' is the next state, alpha is the learning rate, gamma is the discount factor, and max_a'(Q(s',a')) represents the maximum Q-value over all possible actions an' in the next state s'.

While both SARSA and Q-learning are effective reinforcement learning algorithms, they may not necessarily be the best solutions for all problems. In the case of the Connect Four, these algorithms may not perform well due to the large state space and the complexity of the game. Connect Four has a huge number of possible positions, and it can be difficult for these algorithms to explore and learn the optimal policy. In addition, the algorithms may suffer from the problem of overfitting, where they memorize specific moves rather than learning general patterns. To overcome these challenges, more advanced techniques such as deep reinforcement learning may be required.

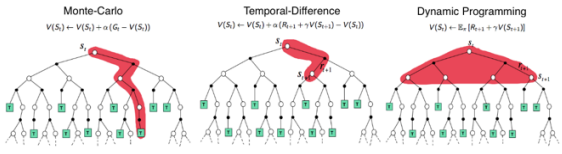

Figure 2 : Comparison of diagrams of Monte-Carlo, Temporal-Difference learning, and Dynamic Programming for state value functions [4]

Deep Q-learning has several advantages over traditional Q-learning, including the ability to handle high-dimensional state spaces, improved generalization, and the ability to learn complex non-linear relationships between states and actions.

The deep-Q neural network is trained by minimizing the difference between the predicted Q-value and the actual Q-value obtained from experience. This is done using a variant of stochastic gradient descent known as mini-batch gradient descent, where the network parameters are updated using a small batch of randomly sampled experiences.

Deep Q-Networks (DQN) introduce two important mechanisms: experience replay and a periodically updated target network [5]. Experience replay stores past experience in a replay buffer, from which samples are randomly drawn during training. This reduces the correlation between consecutive samples and improves the efficiency of the learning process.

Moreover, DQN introduces a separate target network that is periodically updated with the Q network's weights. This target network is used to generate the target Q-values in the Q-learning update rule, which reduces the feedback loop and makes the training more stable.

Additionally, deep Q-learning can be combined with other deep learning techniques such as convolutional neural networks (CNNs) to handle visual input, making it particularly useful for tasks such as image classification and video games.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 3 : Algorithm for DQN with experience replay and occasionally frozen optimization target [5]

These characteristics make DQN a powerful and effective algorithm for solving a wide range of reinforcement learning problems, particularly those with large state and action spaces. In that end we now introduce our implementation.

5. Deep-Q-learning implementation

First, we would train a Deep Q-Learning agent using a Multi-Layer Perceptron (MLP) to approximate the Q-values for the game of Connect Four. The agent would play against an opponent that makes random moves, which would serve as a baseline for the agent's performance. The MLP would take as input a vector representation of the board state and output Q-values for each possible action. The agent would use the Experience Replay and Target Network mechanisms of DQN to improve its performance over time.

Then, we would use a Convolutional Neural Network (CNN) as the function approximator for the Q-values, taking the board as a visual input. The CNN based agent would be trained using the same DQN algorithm (as the MLP agent), but now the input would be an image of the game board instead of a vector representation. This would allow the agent to learn more complex features from the game board and potentially improve its performance over the MLP-based agent.

Finally, after training both agents, we could evaluate their performance by making them play against each other. This would provide an interesting comparison between the two agents, as well as an indication of how well the CNN-based agent can learn from visual input compared to the MLP-based agent.

5.1. « Player » class

The "Player" class encapsulates the logic of an agent that learns to play to Connect Four by training a neural network to predict Q-values for different actions in different states. The training process involves sampling transitions from a replay memory, computing the target Q-values, and minimizing the error between the predicted Q-values and the target Q-values. The value of epsilon is gradually reduced during training to encourage the agent to exploit its learned policy.

The constructor method initializes the various parameters of the player such as the initial value of epsilon, minimum value of epsilon, epsilon decay rate, gamma (discount factor), learning rate, batch size, replay memory, neural network model, target model, optimizer and loss function.

The "get_action" method selects the action to be taken by the player in the current state. It chooses a random action with probability epsilon and chooses the action with the highest Q-value predicted by the model with probability (1-epsilon).

The "remember" method stores the current state, action, reward, next state, and termination flag in the replay memory.

The "replay" method randomly samples a batch of transitions from the replay memory and performs a gradient descent step to minimize the mean squared error between the predicted Q-values and the target Q-values.

The "update_target_model" method copies the weights of the current neural network model to the target model.

The "decay_epsilon" method reduces the value of epsilon by multiplying it with the decay rate, subject to a minimum value of epsilon.

5.2. « MLP » and « CNN » classes

The "MLP" class defines a multi-layer perceptron model with 3 fully connected layers. The constructor method initializes the layers with the specified input and output dimensions. The "forward" method performs a forward pass through the network by applying the linear transformations and activation functions defined in the layers.

The "CNN" class defines a convolutional neural network model with 2 convolutional layers and 2 fully connected layers. The constructor method initializes the layers with the specified input and output dimensions. The "forward" method performs a forward pass through the network by applying the convolutional filters, pooling, and activation functions defined in the layers.

5.3. Function to train the players.

The method " function training_player " trains a player using Deep Q-Learning to play a game against a random player in a given environment.

Here's a breakdown of the process:

(1) A message is printed to inform the user that the training process has started and the number of episodes for training.

(2) The environment is defined.

(3) The initial state is set by resetting the environment.

(4) A loop is started to iterate over the number of episodes specified.

(4.1) The environment is reset at the beginning of each episode to start a new game.

(4.2) The termination variable is set to False to keep the game going until it's finished.

(4.3) A loop is started to keep the game going until it's finished.

(4.3.1) The player gets an action to take based on the current state.

(4.3.2) The action is taken in the environment and the new observation, reward, termination, and info are returned.

(4.3.3) The current state, action, a reward, observation, and termination are remembered by the player.

(4.3.4) The current state is set to the new observation.

(4.3.5) The player's replay buffer is used to train the player.

(4.4) The target model used for training is updated.

(4.5) The player's epsilon value is decayed.

(4.6) After every 100 episodes, the current number of wins is calculated.

(4.7) The episode number and the number of wins are printed.

(5) After all the episodes are completed, a message is printed to inform the user that the training process is complete.

(6) The trained player is returned.

5.4. Function to simulate games.

The method "simulate_game " simulates a number of Connect Four games between two players, and reports the percentage of games won by each player.

Here's a breakdown of the process:

(1) The function starts by printing out the number of games it will run. It then initializes two variables to keep track of the number of wins for each player.

(2) It then enters a loop that runs for the specified number of games. In each iteration of the loop, a new Connect Four game board is initialized by calling env.reset(). The variable state is assigned to the current state of the game board, and the variable termination is set to False.

(2.1) The code then enters another loop that runs until the end of the game. Within this loop, the variable state is set to the last state of the game board by calling env.last(). Depending on which player's turn it is, the code determines which player object to use (Player1 or Player2) and calls the get_action() method on that player object, passing in the current state of the game board as an argument. The resulting action is then executed on the game board by calling env.step(action), which returns the updated game board, the reward received for the action, whether or not the game has ended, and any other information.

(2.2) After the game has ended, the winner is determined by looking at the current player (env.current_player). The win count variable is incremented.

(3) After all the games have been played, the win percentages for each player is calculated and printed out.

6. Results

We creates two players (one MLP and one CNN), trains them using the training_player function with 5000 episodes each, and then runs a simulation of 1000 games between the two trained players using the simulate_game function :

(1) Player_MLP = training_player(Player(MLP()), connect_four_v3(), 5000)

(2) Player_CNN = training_player(Player(CNN()), connect_four_v3(), 5000)

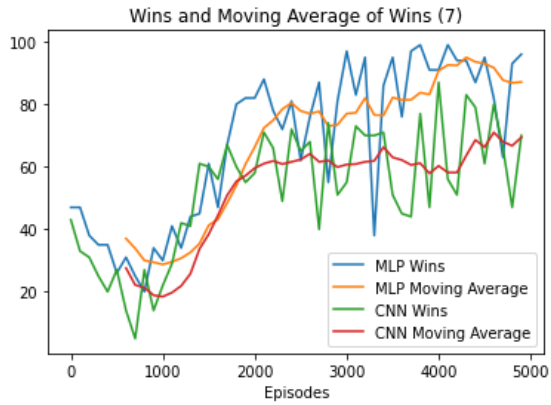

Figure 4 : Wins and Moving average of Wins during training for MLP player and CNN player.

The results show the performance of the MLP-based and CNN-based agents as they were trained and played against a random baseline opponent in the game of Connect Four. The performance metric used is the number of wins by each agent after playing a certain number of episodes.

The MLP-based agent's performance improved over time, with a peak of 99 wins after 4900 episodes. However, there were some fluctuations in performance, with some episodes showing a decrease in wins compared to the previous episode. Overall, the agent was able to learn and improve its strategy against the random opponent.

On the other hand, the CNN-based agent's performance was more erratic, with a peak of 87 wins after 4400 episodes. The agent's performance seemed to fluctuate more compared to the MLP-based agent, with some episodes showing a large decrease in wins compared to the previous episode. This may be due to the complexity of the CNN-based agent's architecture and the high-dimensional input it receives, making it harder to learn and generalize from the training data.

When comparing the two agents' performances against each other, but the results are not clearly decisive.

After the first training it is clear that the MLP-based agent outperformed the CNN-based agent, achieving a higher number of wins after training (see figure 5). This could be because the MLP is better suited to learn from the vector representation of the board state, while the CNN may not be able to extract useful features from the visual input.

Training 1

Configuration	Player 1	Player 2
1 : MLP - CNN	96.70%	3.30%
2 : CNN – MLP	8.40%	91.60%
3 : CNN - CNN	92.40%	7.60%
4 : MLP - MLP	99.40%	0.60%

Figure 5 : % of wins for 1000 games for the four possible games configuration after first training.

However, we then reinitialize both players, re-trained them in the same manner as training one, and re-run the games. The learning during training is similar as during training 1 (same plot as figure 4) but the games outputs are different.

Training 2

Configuration	Player 1	Player 2
1 : MLP - CNN	1.90%	98.1%
2 : CNN – MLP	94.40%	5.60%
3 : CNN - CNN	88.10%	11.90%
4 : MLP - MLP	98.80%	1.20%

Figure 6 : % of wins for 1000 games for the four possible games configuration after second training.

It is important to note that the CNN-based agent is trained using visual input (the game board image), which is more complex than the vector representation used by the MLP-based agent. This suggests that the CNN-based agent may have the potential to learn more complex strategies than the MLP-based agent if trained for a longer period or using more sophisticated architectures.

In both training the last two configurations (3 and 4) show that the player who goes first has a significant advantage, regardless of the strategy used by both players. This is a common finding in Connect Four and other similar games, as the player who goes first has more opportunities to create a winning position.

Overall, these results highlight the importance of selecting the right neural network architecture and training strategy for a given problem using Deep-Q-learning, as well as the impact of the starting position on the outcome of the game.

7. Conclusion

In conclusion, traditional dynamic programming and Monte Carlo methods can be effective for games with small search spaces, but they become impractical for games like Connect Four with a large search space. TD learning, a combination of DP and Monte Carlo methods, is an RL algorithm that can further improve the convergence speed.

However, advanced techniques like deep reinforcement learning may be necessary for games like Connect Four, which have a huge number of possible positions and complex decision-making processes. Deep Q-learning, in particular, is an efficient algorithm that combines neural networks and Q-learning to learn the optimal policy.

The results of our simulation show that the MLP-based agent can sometimes outperform the CNN-based agent in terms of the number of wins after training (see figure 5). However, the CNN-based agent is trained using visual input, which suggests it has the potential to learn more complex strategies (see figure 6).

Finally these findings demonstrate the potential of RL techniques to solve complex games like Connect Four and the importance of choosing the appropriate algorithm and architecture for the problem at hand, and the randomness during and at the end of training (which would explain the different outputs for training 1 and training 2).

8. References

[1] Vandewiele, Gilles. “Creating the (Nearly) Perfect Connect-Four Bot with Limited Move Time and File Size.” Medium, Towards Data Science, Nov 28, 2017.

[2] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction; 2nd Edition. 2017.

[3] Gelly et al. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions https://sites.ualberta.ca/~szepesva/papers/CACM-MCTS.pdf.

[4] David Silver's RL course lecture 4: "Model-Free Prediction"

[5] Volodymyr Mnih, et al. Human-level control through deep reinforcement learning. Nature 518.7540 (2015): 529.