Gym Environment and DQN Training for an Advanced Connect4 Variant

Andrea Borghesi - Matr. 00001060539

Autonomous and Adaptive Systems M

Abstract. The objective of this project is to develop a gym-like environment for an advanced variation of Connect4, called *4 in a Row - Evolution*, and to design and train a Deep Q-Network (DQN) model to play the game. The trained model was evaluated against a random opponent augmented with a heuristic strategy. Initially, the untrained model achieved approximately a 20% win rate and a 70% loss rate against the opponent. However, after around 100,000 training episodes, our best-performing model - a Double Dueling DQN with two convolutional layers and approximately 950,000 parameters - attained a 82% win rate and a 10% loss rate. These results demonstrate a promising initial performance, although a human player would likely achieve a higher win rate against this specific opponent.

1 Introduction

Connect4 is a well-known board game where two players compete to place four coins in a row - vertically, horizontally, or diagonally - on a 6×7 board. The game has already been solved using various approaches, and the resulting trained models are so effective that they never lose against a human player, who can at best achieve a tie. Designing a model to play and win at Connect4 is not a trivial task. The game board consists of only 7 columns and 6 rows, with players taking turns to drop one coin per round. The game ends when one player forms a row of four coins. Despite the relatively small board size, the number of possible game states remains manageable, and the action space is limited to just 7 possible moves (one for each column).

In this study, we explore the potential of Deep Q-Network (DQN) models for a more complex variation of Connect4, called "4 in a Row - Evolution". This new version has some rules of the original game, such as the 6×7 board and the coin-drop mechanism. However, it introduces additional rules that significantly increase the complexity of the game. The new rules are as follows:

- At the beginning of each round, both players simultaneously choose 3 columns in which to drop their coins.
- The moves are then executed alternately, and if a chosen column is already full, the coin will fall into the nearest available column.

– The game ends when the board is completely filled, and the player with the most completed four-in-a-row lines wins.

These new mechanics increase both the state and action spaces of the game:

– Each player now has 343 possible action combinations per round.
– The total number of possible legal states is estimated to be approximately 10^{16}, as the game does not end immediately when a player forms four coins in a row.

As will be further discussed in Section 3, tackling this larger problem required experimenting with various DQN-based models. Initially, we implemented a standard "vanilla" DQN model. Later, we introduced enhanced versions incorporating well-known DQN improvements, such as Double DQN, Dueling DQN, and a combined version known as Double Dueling DQN. To leverage the spatial structure of the board, we also experimented with deeper neural network architectures that included convolutional layers.

2 Environment

To simulate a game episode, a custom Gym-like environment has been developed in Python (see Appendix 7.2). This environment maintains the game's state and provides the methods required for training. The primary methods include:

– **reset**: Initializes a new game instance.
– **step**: Executes the player's action and the opponent's move. The opponent plays first and follows a combination of random and heuristic logic (see Section 4).
– **_next_observation**: Returns the current state of the map as a matrix, where cells contain values of 1 for yellow coins, -1 for red coins, and 0 for empty spaces, as shown in Figure 1.

```
[[ 0  0  0  1  0  0  0]
 [ 0  0  1 -1  0  0  0]
 [ 1  0 -1 -1  0  0 -1]
 [ 1  0 -1 -1  0  0  1]
 [ 1  1 -1  1  0 -1 -1]
 [-1  1  1  1 -1  1 -1]]
```

Fig. 1. Example of a printed map state

3 Models

Due to the large state space (approximately 10^{16}), traditional reinforcement learning techniques such as Monte Carlo methods or Q-learning are unfeasible. Consequently, it is necessary to adopt an approach like Deep Q-Network (DQN), which leverages the power of deep learning.

3.1 DQN

Deep Q-Network (DQN)(3) extends Q-learning to handle problems too large for tabular methods. The update step is performed as follows:

$$\begin{array}{cc} \text{if } t \text{ is terminal} & Q(s_t, a_t) = r_t + \gamma \max_a Q_{target}(s_{t+1}, a) \\ \text{else} & Q(s_t, a_t) = r_t \end{array}$$

The Q-values are computed using the current model, whereas the Q_{target}-values are obtained from a target model. The target model's weights are periodically updated to match those of the current model after a predefined number of episodes, known as the **target update interval**, which is a hyperparameter.

Each step of the episode is stored in a fixed-size **replay buffer**, and training is performed by sampling mini-batches from it.

3.2 DQN with CNN

To leverage the spatial information of the map, the DQN approach can incorporate convolutional layers(1). These layers can help identify patterns, such as three coins in a row, which could assist in decision-making. However, due to the small map size, deploying large kernels or multiple convolutional layers is impractical, as the local receptive field would exceed the map dimensions.

3.3 Double Deep Q-Network (Double DQN)

Double Deep Q-Learning(2) (Double DQN) improves upon the standard Deep Q-Learning (DQN) by addressing the issue of **overestimation bias** in action-value estimates.
In the standard DQN, the same Q_{target}-network is used to both select and evaluate actions when computing the target Q-value. This leads to an overestimation of Q-values because the maximum operator tends to prefer overestimated values.

Double DQN decouples the action selection and evaluation steps to reduce this overestimation bias:

- The **current Q-network** selects the action with the highest Q-value:

$$a^* = \arg\max_{a'} Q(s', a')$$

– The **target Q-network** evaluates the Q-value of this selected action:

$$y = r + \gamma Q_{target}(s', a^*)$$

This separation ensures that the evaluation is more accurate and reduces bias. Also, the policy learned is typically more robust for many environments.

3.4 Dueling Deep Q-Network (Dueling DQN)

Dueling Deep Q-Network(4) (Dueling DQN) improves the standard Deep Q-Learning (DQN) by changing its underlying network architecture.

In the traditional DQN, a single neural network is used to approximate the Q-value for each action in a given state. This approach treats all state-action pairs equally, even when some actions may have little impact on learning in certain states.

Dueling DQN addresses this limitation by separating the Q-value estimation into two components:

– **State Value Function (V):** Represents how good it is to be in a given state, independent of the action taken.
– **Action Advantage Function (A):** Measures the advantage of choosing a specific action over other available actions in the state.

The modified network structure is given by:

$$Q(s, a) = V(s) + \left(A(s, a) - \max_{a'} A(s, a') \right)$$

To recap:

– It estimates the overall value of being in state .
– It captures the relative benefit of selecting action compared to other actions.
– Subtracting the maximum (or mean) advantage ensures stable Q-values.

3.5 Double Dueling DQN

Dueling DQN can be combined with Double DQN to leverage both architectural improvements and bias reduction, resulting in a more robust reinforcement learning algorithm.

4 Training

Initially, episodes were simulated by having the model play against a completely **random opponent**. The model quickly learned a simple strategy that allowed it to win almost every game. It discovered that consistently dropping coins in the middle column was sufficient to defeat a random opponent. However, such a

simplistic and predictable strategy proved ineffective against a human player. To prevent this obvious and ineffective solution, the random opponent was enhanced with a simple yet effective **heuristic**: it would drop a coin in a column where three of the same color were already present, increasing the probability of either scoring or blocking the opponent from achieving a vertical four-in-a-row.

The previously trained model proved completely ineffective against this new heuristic-based opponent, making it considerably more challenging to train a new model capable of achieving a good score.

Instead of performing a training step after each action, as typically described in Q-learning literature, it has been decided to wait until the end of the episode to obtain a final reward (1 for a win and -1 for a loss). Although this approach aligns more closely with Monte Carlo methods, the short duration of the game (each episode lasting exactly 7 rounds) made it practical to store state-action-reward tuples in the replay buffer after the episode concluded.

Eventually, multiple training steps were executed at the end of each episode. During each training step, a batch of 64 or 128 tuples was sampled from the replay buffer and used to update the model according to the chosen strategy (DQN or Double DQN).

To enhance model robustness, the opponent's move was randomly determined using either the target model, the current model, or the *best model* (the model that achieved the highest evaluation score so far).

4.1 Exploration

During training, a standard ϵ-greedy approach was implemented. The exploration level was regulated by the following hyperparameters:

- *eps_start*: Initial value of ϵ.
- *eps_end*: Final value of ϵ.[1]
- *min_episodes_before_eps_update*: Initially, ϵ remains high to encourage exploration. After the specified number of episodes, ϵ begins to decrease linearly.

To further promote exploration, an additional hyperparameter called *max_random_skips* allows the current player to randomly select one of the actions with the highest Q-value instead of always choosing the current best action.

4.2 Evaluation

Evaluation is conducted every n episodes by playing 500 games against the random-heuristic opponent to track the model's improvement. In this way, it has been possible to store the weights of the model that achieved the highest score.

[1] It is generally recommended to set the final ϵ value between 0.01 and 0.05.

5 Results Overview

Once training was completed, 10,000 games were simulated between the best model and the random-heuristic opponent to obtain a more reliable performance evaluation. Similar simulations were conducted using a completely random opponent and an untrained model to establish a benchmark. The summarized results are presented in Table 1 below:

Table 1. Comparison of Initial and Final Scores

Scenario	**Win Rate**	**Tie Rate**	**Loss Rate**
Untrained vs Random Opponent	29.0%	13.6%	57.4%
Untrained vs Random Opponent (Heuristic)	19.1%	9.9%	71.0%
Best vs Random Opponent	92.0%	4.1%	3.9%
Best vs Random Opponent (Heuristic)	82.1%	7.7%	10.2%

Through multiple trials, the best-performing model contained 950,000 parameters; however, a smaller version with 460,000 parameters achieved comparable results. The model architecture is detailed in Appendix 7.1. The model leverages a Double Dueling DQN approach and was trained for 80,000 episodes over several hours on Google Colab.

Finally, Figure 2 illustrates how the evaluation score increases rapidly during the early stages of training, then gradually slows down as it approaches the maximum value, eventually stabilizing with minor oscillations.

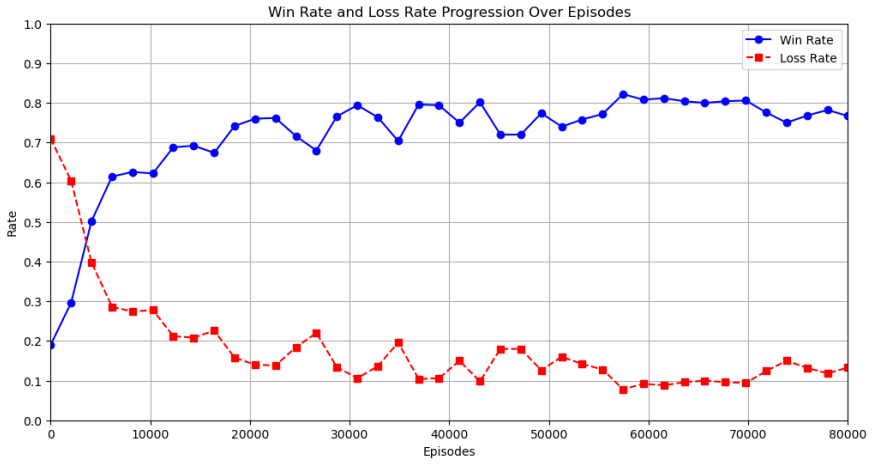

Fig. 2. Win and loss rate progression from episode 0 to 80,000

6 Conclusions

The obtained results could be further improved by applying more advanced reinforcement learning techniques and utilizing greater computational power (this study was constrained by Colab's limitations). There are still many network settings and hyperparameters to try, but testing them would take much more time due to the mentioned limitations.

Even with these limitations, the results show that reinforcement learning can be very effective when used with deep learning techniques.

7 Appendix

7.1 Model architecture

The model architecture is plotted in Figure 3 in the next page.

7.2 Code

All the code is available at https://github.com/Borgo99/Gym-Environment-and-DQN-Training-for-an-Advanced-Connect4-Variant .

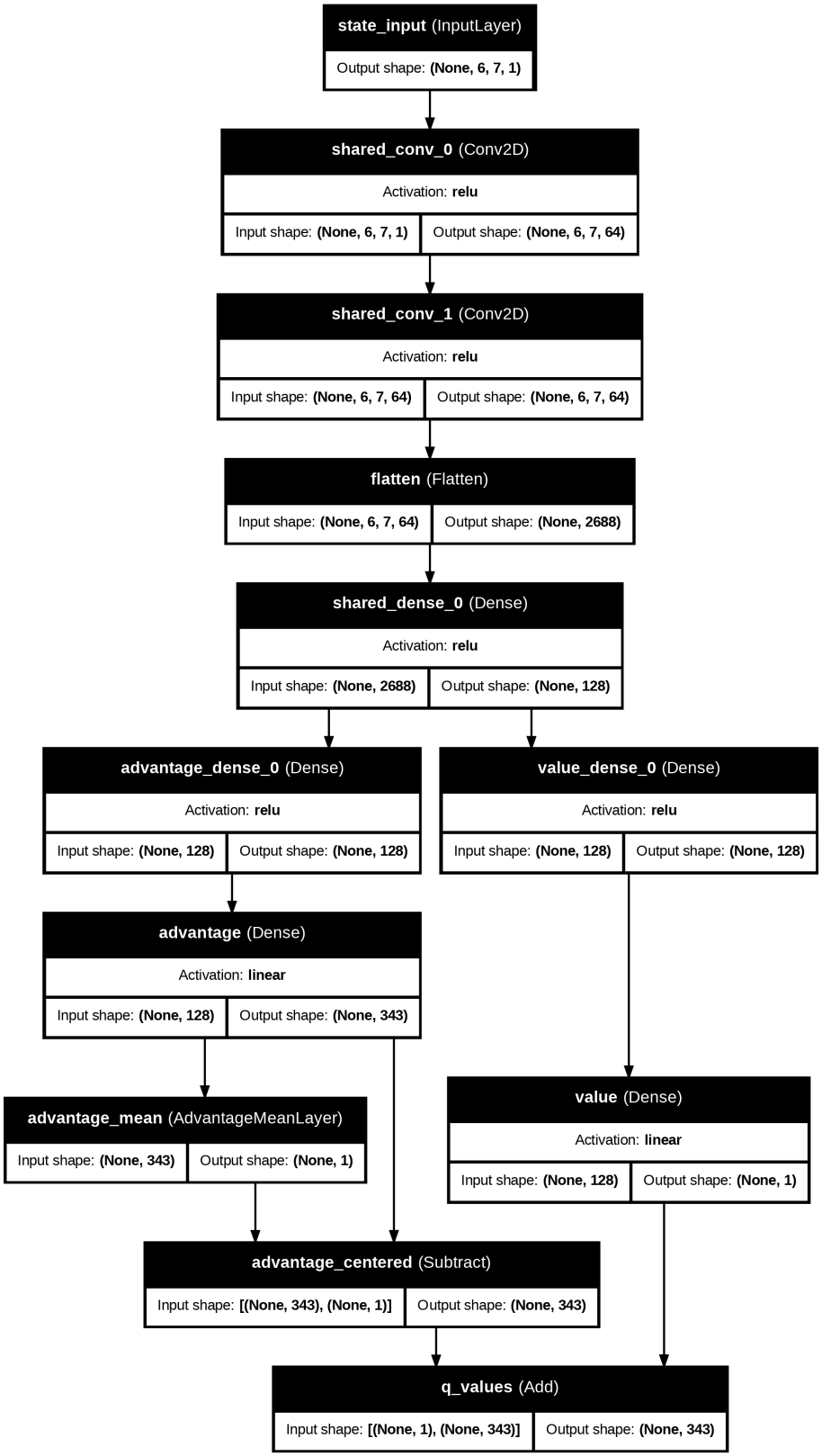

Fig. 3. Architecture of the model that achieved the highest score

Bibliography

[1] Codebox: Connect4 - codebox (2020), https://codebox.net/pages/connect4, accessed: 2025-01-27

[2] van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning (2015), https://arxiv.org/abs/1509.06461

[3] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning (2013), https://arxiv.org/abs/1312.5602

[4] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., de Freitas, N.: Dueling network architectures for deep reinforcement learning (2016), https://arxiv.org/abs/1511.06581