COMP 579 - Final Project Report

Can Akin - 260891162 **Joey Chuang - 260893876** **Steven Koniaev - 261028242**

Department of Computer Science
McGill University
845 Rue Sherbrooke O, Montréal, QC H3A 0G4
{can.akin, ching-i.chuang, steven.koniaev}@mcgill.ca

Abstract

In this study, we investigated the performance of two reinforcement learning models in multi-agent sparse reward environments. We examined games like tic-tac-toe, Connect 4, and Chess. With our limited compute we focused on Connect 4 using Deep Q-Networks and Actor-Critic algorithms. While Connect 4 is simpler than Chess it does have interesting tactics to explore and thus makes it a great environment for investigating these algorithms. We implemented both models from scratch along with several variations. Our results demonstrated that both algorithms can learn these games given sufficient training time, although the efficiency of training varies significantly with different modifications to the algorithms. Our findings show that in a multi-agent setting, the first agent always has an advantage when training. We compared our algorithms against random bots and then against each other. We demonstrate that there is an increase in performance when updating an agent based on what we define as key states and allows for faster training. The project was implemented using Python and relied on the NumPy, PyTorch, and Matplotlib packages. We concluded that these algorithms can effectively learn these games, that the first player always has an advantage due to the nature of how these algorithms learn, and there are some strategies that could be explored further to propagate the key state actions pair values quicker to other states.

1 Introduction

The exploration of reinforcement learning in game environments provides a rich context for advancing artificial intelligence techniques, particularly in games that blend strategic depth with computational tractability. We focus on Connect Four, a classic game which involves developing a strategy to guarantee a win. We use the PettingZoo package, which offers an API for interacting with multi-agent environments. This project seeks to diverge from traditional approaches to solving these environments, such as Monte Carlo Tree Search (MCTS), which, despite its success in various game environments is very computationally expensive and slow. Our focus will be on investigating the potential of classic reinforcement learning algorithms such as Deep Q Learning and Actor-Critic methods to look at how these classic algorithms perform on challenging sparse reward environments.

To compare these two models, we have implemented the Deep Q Network and Actor-Critic models using Python and packages such as NumPy, PyTorch, and Matplotlib. Our methodology involved several experiments. Initially, each model was trained against random agents; subsequently, they trained against copies of themselves; and finally, we tried different comparisons and further training between agents and algorithms by introducing a buffer for Key States. Our findings confirm that both algorithms can effectively learn the dynamics of these games. Additionally, we observed a consistent advantage for the first player, which can be attributed to the nature of these algorithms. Finally, we found that using key states helps propagating back Q values throughout the network as new strategies are discovered.

Preprint. Under review.

2 Background

2.1 Environment Details

The PettingZoo library [1], is a comprehensive library for multi-agent reinforcement learning which provides an API for Connect Four. Connect Four is a 2-player turn based game, where players must connect four of their tokens vertically, horizontally or diagonally. The players drop their respective token in a column of a standing grid, where each token will fall until it reaches the bottom of the column or reaches an existing token. Players cannot place a token in a full column, and the game ends when either a player has made a sequence of 4 tokens, or when all 7 columns have been filled.

The observation shape is a [2,6,7] tensor, which encodes the states of the board for both players. The action space is discrete from values of 0 to 6, one for each columns to drop a token in. The Petting Zoo API also provides an action mask for selecting an action and instantiates and destroys agents as the environment begins and ends.

For each step, the agent receives a reward of 0 until a termination occurs. The winning player gets a reward of 1, the losing player a reward of -1 and a reward of 0 if there is a draw.

2.2 Algorithms

2.2.1 Deep Q Learning

Using the Bellman equation for Markov Decision Processes we have the following criterion:

$$MSE(Q(S_t, A), R_{t+1} + \gamma argmax_a Q(S_{t+1}, a))$$

Where the Q function will be approximated with a deep neural network. We use a Convolutional Neural Network to leverage the grid like state space. Here, the DQN starts with two convolutional layers that extract features from the input, each using a 2x2 filter. The output from these layers is then flattened into a one-dimensional vector. This vector is processed by a series of linear layers: the first transforms the flattened features into 128 outputs, a ReLU activation function introduces non-linearity, and a final linear layer produces 7 outputs. It is typical for models in reinforcement learning, where the model needs to understand spatial hierarchies and then make decisions based on learned features.

We used two different versions of DQN for sample efficiency.

First, we used a buffer of states which are sampled with a batch size of 128 per episode. This allows for smoother and more consistent training by the loss being averaged over the batch.

Next, we implemented parts of RainbowDQN to add even greater sample efficiency while trying to keep our compute costs the same, specifically we implemented a priority replay buffer and noisy linear layers.

We modified the final linear layers of the neural network to increase the exploration rate of the network. Exploration is crucial in reinforcement learning, as the agent must try various actions to gather information. Rainbow DQN injects noise into the network during training to facilitate exploration. We followed the original paper's approach by implementing a new layer by sampling from two Gaussian distributions [2].

We also modified our original buffer to prioritize certain experiences. This technique, known as Prioritized Experience Replay (PER), assigns weights to experiences within the buffer based on their TD-error. The magnitude of a TD-error reflects the "surprise" associated with a transition. Transitions with larger absolute TD-errors are considered more informative and are therefore replayed more frequently. The weights themselves are updated based on the TD-error, with smaller errors resulting in smaller adjustments to the replay probability, and larger errors leading to more significant changes.[3].

2.2.2 Actor Critic

Actor Critic works by having two functions. An actor that picks actions, and a value function which tells the actor how good a certain action was.

We first compute the TD-Error using the Bellman Equation as:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

Using the policy gradient theorem of the expected return J, we update each network as follows:

$$MSE(V_{pred}(s'), V_{actual}(s'))$$

Where s' is the next state.

Using the Advantage defined as:

$$A(s, a) = r + \gamma V(S') - V(S)$$

We update with the criterion:

$$\nabla J \approx \frac{1}{N} \sum_{i}^{N} \nabla log\pi(a_i|s_i) A(s_i, a_i)$$

The Actor Critic model here is a neural network structured to output action probabilities for reinforcement learning. It consists of two convolutional layers, each with a 2x2 filter, designed to extract spatial features from inputs. These layers are followed by a flattening process that prepares the data for a series of linear transformations. The network processes the flattened data through linear layers, including a ReLU for non-linearity, and ends with a Softmax layer that converts the outputs into a probability distribution over actions.

2.2.3 Key State Buffer

We introduced an additional buffer to see how it affects the learning of the algorithms. This buffer is also implemented as a priority buffer, however, we only store states when the reward is either +1 or -1. Because the environment has sparse rewards we hope that this will help propagate the reward signal to the starting states from the trajectories chosen.

3 Related Work

Games have long served as benchmarks for applying reinforcement learning algorithms, especially in multi-agent environments that interact with each other. The first major breakthrough in reinforcement learning with games is the first paper on Deep Q Learning [4] by Mnih, where researchers used neural networks to approximate Q values to play Atari games. This has been followed by the work of Ryan Lowe and Yi Wu [5], which introduced Multi-Agent Actor-Critic methods that are used in this study.

Additionally, the strategy of training agents against themselves and utilizing past experiences has been thoroughly investigated by the team at Google DeepMind, during the development of AlphaGo, who used Deep Monte Carlo Search Trees [6]. Although these methods are very successfully we wanted to explore classical algorithms on classical board games without explicit search and with less computational demands.

As we have observed in this project, initial advantage plays a significant role in the performance of these models. This has been also explored by Balduzzi [7], whose research found that the initial mover's advantage is often critical in zero-sum games. Balduzzi worked on this facet of game theory where he studied the symmetry and balance within games to anticipate outcomes based on player sequence, thus providing empirical backing for our findings.

Our project provides an analysis of different algorithms and an attempt to quicken training by adding a buffer specifically for key states in sparse reward environments.

4 Methodology

We test 3 different algorithms, DQN, Actor Critic, and our Improved DQN. For each algorithm we create two agents for each player. We first test against playing against a random bot and see how quickly we are able to consistently win. Next, we take the two agents and play them against each other for 50,000 episodes. Next, we try different variations of either trying to adapt to the stronger players strategy or retraining the weaker side (player 2) with an additional key state buffer (also 50,000). We want to see if the key state buffer will allow to mitigate some of the first player advantage and see how the agents grow and change to counteract each others development.

5 Experiments

5.0.1 Playing a Random Agent

All agents defeat their random agent counterpart as they quickly learn the win condition. The Appendix shows graphs for DQN and our Advanced DQN. Interestingly, the Advanced DQN loses at first several times, as the noisy linear layers encourage initial exploration.

5.0.2 Two Agents Against Each Other

In this experiment we put two agents against each other and see how they evolve over time.

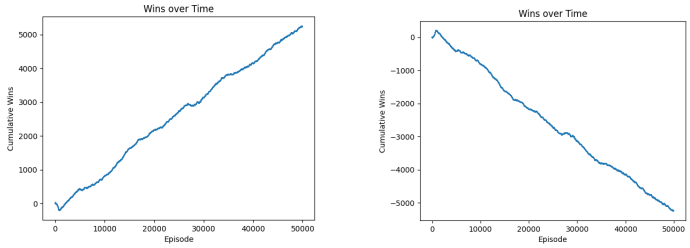


Figure 1: (a) Player 1 (Adv DQN) (b) Player 2 (Adv DQN)

This graph the training of two Advanced DQN agents playing against each other (See Appendix for more).

These results makes sense to us for two reasons. Connect Four is a winning game for player 1 so the first agent should always have some advantage and that the second agent must learn to play around the first, while the first can always take the initiative. It seems that over the course of training the second agent eventually adapts to a strategy of the first, once this happens the first agent tries to change its strategy and successfully continues winning, this is suggested by the noisiness of the graphs.

5.0.3 Adapting to Player 1

If we freeze Agent 1 temporarily, how quickly can the second player adapt to the first?

We find that the number of decisive games quickly drops down to around 500, meaning our agent is able to quickly learn to exploit the weaknesses of the first if we stop training (See Appendix).

5.1 Agent with Key State Buffer

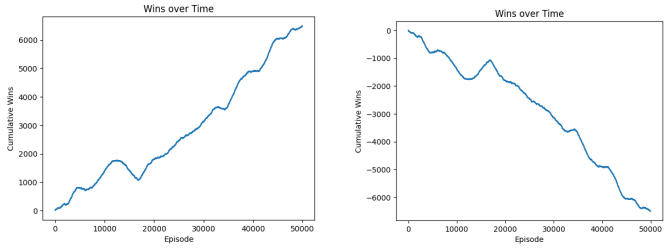


Figure 2: (a) Player 1 (b) Player 2

In order to train the second player quicker we give it a Key State Buffer to see if the additional states will aid in helping more consistently win. We find that the cumulative rewards become a lot more noisy but still lean in the favor of the first Player. Once again this result is expected because Connect Four should be a winning game for Player 1 (although the agents should not be playing perfectly with only 50,000 episodes).

6 Conclusion

To conclude, the experiments conducted in our project reveals that both the Deep Q Network and Actor-Critic models have effectively mastered the dynamics of Connect Four. We observed a significant advantage for the first player, who benefits from the initiative to explore new states at the beginning. Importantly, the addition of a buffer for key states has proven beneficial, facilitating the propagation of Q values across the network and enabling the discovery of new strategies.

Moving forward, we plan to extend these techniques to more complex and sparser game environments, such as chess. We also intend to acquire the necessary computational resources to support more sophisticated models. These advancements are expected to significantly accelerate our progress, pushing the boundaries of what is achievable with classic reinforcement learning algorithms in strategic game scenarios.

7 References

References

[1] *PettingZoo*. Farama. Available at: `https://pettingzoo.farama.org`. Accessed [2024]

[2] Hessel, Matteo, et al. "Rainbow: Combining improvements in deep reinforcement learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 32. No. 1. 2018.

[3] Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).

[4] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

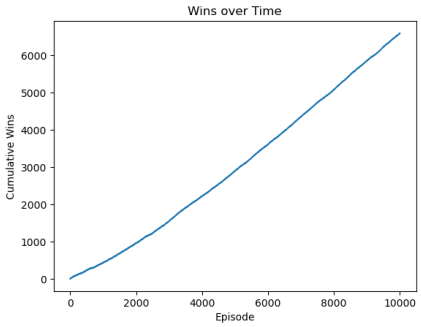
[5] Konda, Vijay, and John Tsitsiklis. "Actor-critic algorithms." Advances in neural information processing systems 12 (1999).

[6] Silver, David, et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search." Nature, vol. 529, no. 7587, 2016, pp. 484–489.

[7] Balduzzi, David, et al. "Open-ended learning in symmetric zero-sum games." International Conference on Machine Learning. PMLR, 2019.

8 Appendix

8.1 Random Agents



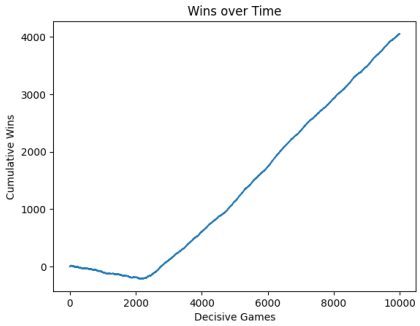
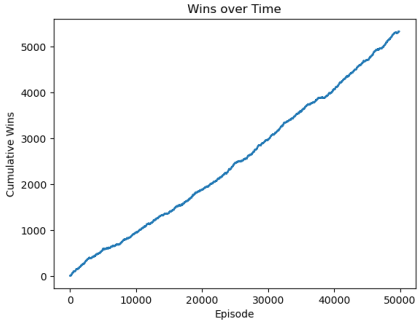


Figure 3: (a) DQN vs Random Player (b) Adv DQN vs Random Player

8.2 Agents Against Each Other



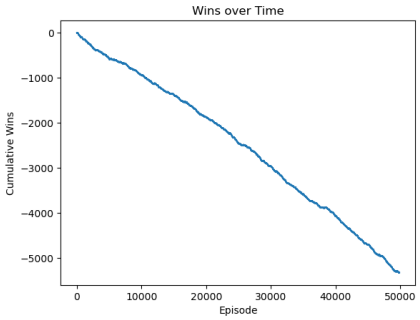
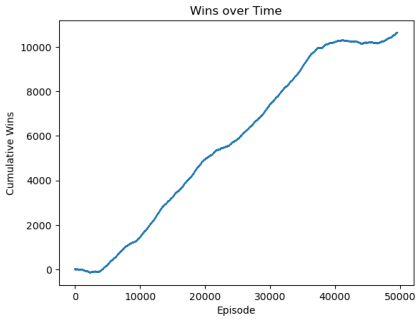


Figure 4: (a) Player 1 A2C (b) Player 2 A2C



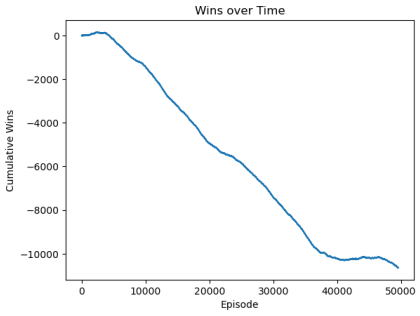


Figure 5: (a) DQN Player 1 (b) DQN Player 2

8.3 Stopping Training on Player 1

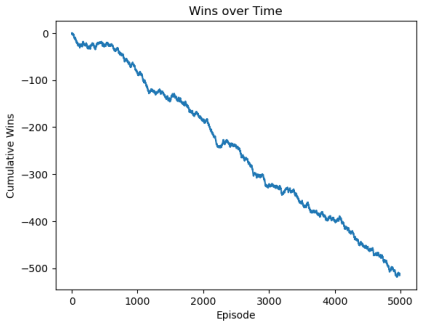


Figure 1: Agent 2 winrate trying to adapt to Agent 1 A2C