Comparing Algorithm Performance in Goofspiel with Adversarial Learning

Greta Freitag
Department of Computer Science
McGill University
Montreal, QC
greta.freitag@mail.mcgill.ca

Abstract

For my project, I decided to compare the performance of different algorithms learned in class in an adversarial environment, in the simple 2 player game of GOPS (also known as Goofspiel). Of interest is how long it takes for one algorithm to become the dominant player, and how the scores of the two players change over time.

1 Introduction

When learning about different algorithms in class and implementing them in the homework, we only ever had our agents operate in neutral environments, where the only changes were the consequence of an agent taking an action. If there was an antagonistic factor to the environment (like in Atari games, for example), the strategy of opponents did not change and could be easily learned over. I became curious how the algorithms we learned would perform against each other, in a simple 2-player game. How would they deal with an evolving environment? How would an agent's policy trained against one opponent perform against an opponent with a different strategy? I was curious about how they would learn to adapt and how flexible different algorithms would be.

2 Background

I should introduce the game my agents will be playing. After reviewing several different 2-player games, I eventually settled on The Game of Pure Strategy(GOPS), also known as Goofspiel. The benefits of this game include a relatively small state space, a clear goal with clearly defined rewards, and an aspect of stochasticity, meaning the agent can't simply learn the perfect strategy.

The rules of the 2-player game of GOPS is as follows:

- A deck is split into suits. One suit (13 cards total) goes to each player and becomes their hand.
- One of the remaining suits is shuffled and placed between the two players. This is the prize deck.
- At the start of every round, the top card of the prize deck is flipped over. This is the current prize card.
- Every round, both players will decide on a card to bid for the prize. They then both reveal their card at the same time.
- The player with the highest bid wins the current prize. In the event of a tie, the value of the prize is split between both players.
- The round now ends, and a new prize card is flipped over. After the prize deck runs out of card, the winner is decided as the player with the most accumulated points.

3 Methodology

For my experiement, I created a custom environment for GOPS, as there is no preexisting Gymnasium environment. States consist of the remaining cards in the prize deck, the cards in both players hand, the current face-up prize card, the previous winner, and whether the game is over. Note that the agent has perfect information about the environment, except for the ordering of the prize deck. While it would be difficult to remember all the details as a human, it is possible to keep track of exactly where each card is at any given time. What makes the game interesting for reinforcement learning is the randomness of the prize card and the bidding choice of the opponent. An agent has to learn to predict what the other agent will play.

The first agent I created was a random one. It is important to see how my other agents perform against the random one, to make sure they are actually learning.

In connection with this, I also wanted an informed, consistent agent to act as a baseline. For this, I chose an agent that performs the best against the random agent. In the game of GOPS, this is the agent which always chooses to bid a card of the same value as the prize card (Ross, 1970). Any learning agent should be able to predict this strategy and learn to bid one higher, making this a good baseline agent.

The first RL algorithm I chose to implement is the Monte-Carlo algorithm. Monte-Carlo is an interesting choice because it requires sampling several fake games based on what the agent knows about the environment, and this meant I would have to make my agent create a model of its opponent in order to generate steps in the game. This brought an interesting twist into an algorithm which otherwise could have simply sampled potential next states in an environment that didn't contain another agent. Since my agents don't have access to each other, I made my monte-carlo algorithm build an approximation of its opponent. Inside my monte-carlo agent, I created a linear function approximator of the opponent, which takes an input of the current state (which both players share), and is trained to output the action the opponent will take (which I round to the closest real number). This approximation is updated whenever the agent takes a real step, based on the error of the predicted next action vs the actual next action. The opponent approximation is then frozen and not updated when we generate our training episodes, but is used for the sake of this generation. I optimized for the hyper-parameters on the baseline agent.

To contrast with MC I also chose to implement a Q-learning agent on GOPS. Since this algorithm has no need to generate future states, it also did not need to create an approximation of the opponent. I predicted that this will be a downside of this algorithm in the environment. But, this is also a much more efficient and faster algorithm than Monte-Carlo, so it is worth comparing its performance to the much more memory-intensive MC algorithm. I also used a linear function approximator for this algorithm. (I attempted an MLP, but it just wasn't working out for me. A good future study.)

As my final algorithm, I also implemented a policy-based algorithm, REINFORCE. I wanted to see how an agent with a neural network policy approximator would compare against ones using a linear function approximator for state value. I also struggled with the other policy gradient algorithm we implemented in class, Actor-Critic, so I wanted to have a second try at implementation.

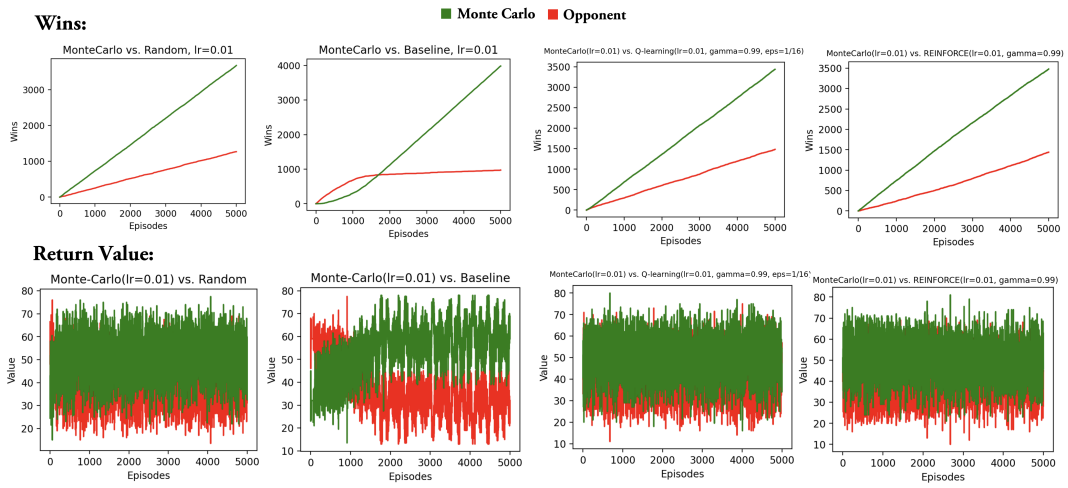
I had all my agents compete against each other in GOPS, where they were given 5000 episodes to figure out the optimum strategy against their opponent. There were 5 tournaments per learning algorithm, and shared parameters like learning rate were set to the focus algorithm's best value (determined over several tests).

4 Experimental Results

I will go over the performance of all the RL agents one by one, and discuss the results.

4.1 Monte-Carlo

Monte-Carlo Tournaments, learning rate of 0.01 over 5000 episodes:

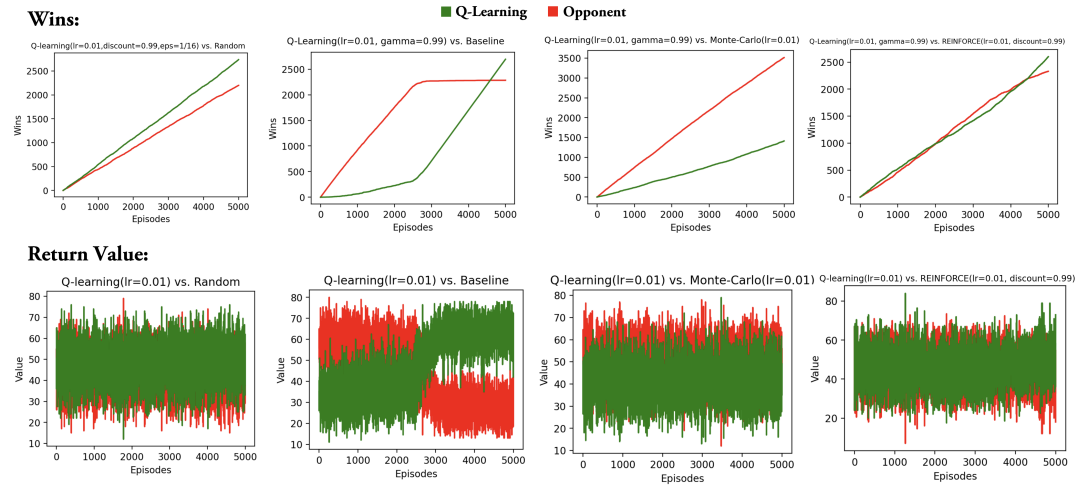


The Monte-Carlo algorithm had by far and away the best performance out of all the ones I implemented. Because of the randomness of the Random agent, and the constant learning of the other agents, it doesn't win all the time, but it consistently more successful and higher scoring. Against the Baseline, which selects cards consistently, MC learns the strategy almost immediately and after the 1000th time step wins almost every time. I think the success of MC here has to do with its time allowance for simulating games. Since there is no time limit on agents for picking moves, MC effectively plays over 5x the amount of games as the other agents at any given time step. This allows it perform much better. Even though it is one of the slowest running algorithms, the consistent 13 time-step episodes certainly gives it a leg up in this game.

It also performed much better against the random agent than the other algorithms, likely because the quality of its approximation of the opponent doesn't matter in the random case, and its simulated games against Random had the same value as the real ones. Again, it effectively got 5x the training in this case.

4.2 Q-Learning

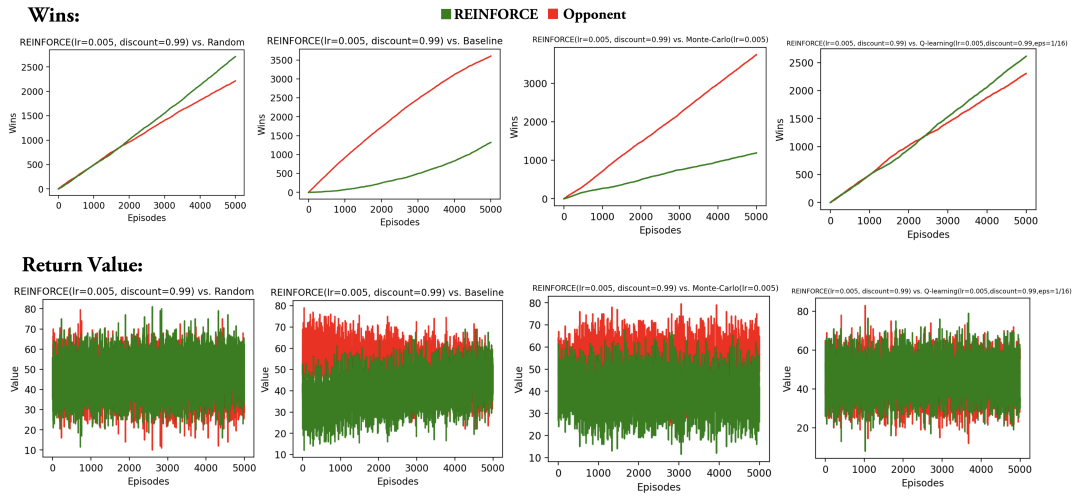
Q-learning Tournaments, learning rate of 0.01, discount factor of 0.99, epsilon of 1/16, over 5000 episodes:



I wanted to include Q-learning to compare to the similar Monte-Carlo algorithm, but it became clear which one worked best in this environment very quickly. Bootstrapping just isn't as effective as learning via experience in an environment where the opponent agents are constantly learning and changing. The Q-values it learns quickly become inaccurate and the speed of policy change in the opponent puts Q-learning at a disadvantage. Against agents like Baseline however, it performs quite well and eventually learns the perfect counter strategy. Interestingly, it is on-par with REINFORCE most of the time, and which one wins comes down to hyperparameters like the learning rate.

4.3 REINFORCE

REINFORCE Tournaments, learning rate of 0.005, discount factor of 0.99, over 5000 episodes:



REINFORCE ended up being my most disappointing algorithm. I don't know if it's the Pytorch Neural network, but it trained extremely slowly. For such a simple game, the number of hidden layers and nodes is minimal and I adjusted parameters consistently, but still the learning was still very slow. I believe it faced similar issues to Q-learning with adapting its policy to the constantly changing policies of its opponents. Furthermore, REINFORCE required an extremely low learning rate, as with higher ones, at a certain point the neural network's parameters blow up and it stops learning. I believe this may be why It stopped overcoming baseline and consistently won half the time at later episodes in the tournaments. If this is not the case, it's possible that it would eventually learn Baseline's strategy, but I limited my agents to 5000 episodes. I got the impression that REINFORCE requires more training time than the other simpler algorithms.

5 Conclusions and Future Work

Since each agent must make a play before a step is completed, the speed of an algorithm has no impact on gameplay, unless there were some timer added for each round. Because of this fact, Monte-Carlo, which generated and trains on several additional episodes per times step has a significant advantage. Therefore, It would be interesting to place some time restrictions on future experiments and limit the number of generated episodes to fit within that time limit. This may increase the ability of the other algorithms to play on par with MC. Additionally, it might be a more fair comparison if I made MC gather its training episodes from actual games played rather than generating fake games. But I still found creating an approximation of the opponent to be an interesting challenge.

Goofspiel is a simple game, and therefore doesn't require deep learning algorithms to train an agent to perform well, but it would be interesting to see how algorithms like DQN would hold up to the agents in the current experiment. I also want to see how policies learned against one opponent translate to competing against other opponents, and I think this would be a good topic for a future experiment.

6 References

Ross, S. M. (1971). Goofspiel — the game of pure strategy. Journal of Applied Probability, 8(03), 621–625. https://doi.org/10.1017/s0021900200035725