AI Calling BS

Victoria Outkin

ABSTRACT:

Games either have complete information such as chess, where all resources and actions are visible, or incomplete information such as poker and the real world, where not all information is available to make a decision. Training artificial intelligence (AI) agents on games with incomplete information has been less successful than with complete information. One of the recent big breakthroughs in games with incomplete information was poker. From the MIT Technology Review, AI capable of playing poker relied on incomplete information, which made it impressive (Knight 2017). In my project I explore another incomplete information game – two player BS – to provide insight into how AI can use information to fill in gaps and learn without knowing the entire picture.

LITERATURE REVIEW AND BACKGROUND:

More and more games can be solved with AI, and there are open source games people can test and train their AI agents with sources such as The Arcade games in Open AI Gym (an open source library with many games available (Ravichandiran 2018). New algorithms to solve games have been made, and there has been a rise in people's interest in reinforcement learning (RL) (Brockman, 2016). With RL, an agent does not know the rules of the game, but rather takes actions, saves experiences, and collects values relating to states and rewards encountered. Many agents use a Markov Decision space, in which the idea

that an action leads to a reward and a new state is saved. Over time, the rewards or favoritism to saving new favorable states decrease, leading to an agent with a policy. For example, an agent can start solving a maze randomly, and as it learns over time how to get through the maze, it is more likely to choose the same path or the same decision rules over and over in the end (since it has learned the behavior over time) (Pettit 2023).

Algorithms and ideas have been introduced for games to be solved more efficiently, so that AI can learn to solve problems in the real world with strategies it used to play games. For example, making an agent go through unique mazes with coins in training runs (instead of being trained repeatedly on the same maze) helps the agent adapt to new mazes and be able to experience new situations (Cobbe, 2019). Ideas such as “sticky actions” in Gotta Learn Fast have a similar goal - helping an agent continue learning over time by adding a bit of randomness by randomly repeating certain actions (Nichol et al, 2018).

Many games are tested in RL with all actions visible such as Cartpole and Pong and algorithms such as Deep Q-Network (DQN) have been developed to help agents learn (Mnih et al.). Research and fascinating science with complete information has been done, such as how neuronlike elements can be used with Cartpole – a game where a pole is balanced (Barto et al.). From playing professional chess to beating the world champion of Go, AI has been able to play complete information games professionally (Haridy, 2017). On the other hand, there has not been as much research with incomplete information, and

Poker was a big game which was a breakthrough to solve - because it uses incomplete information (Knight 2017).

There is less implementation relating to addressing games with incomplete information although research has been done. There have been models and theories addressing partial observability even going back to 1992, such as using multiple hierarchical systems. For example, to solve a maze, instead of the maze being done all the way through by one agent, different agents can be assigned different parts, and then combine their efforts. This makes it more efficient, although more energy for each run might be spent by the agents, when the agents' efforts are combined, as a whole it is faster and saves energy (Dayan, 1992). Different approaches for multi-agents to learn reinforcement learning in partially observed environments have been introduced such as RAIL and DIAL (Reinforced Inter-Agent Learning and Differentiable Inter-Agent Learning) which use multiple agents to complete a task. Games have been made such as MNIST, a game in which two agents communicate with each other to figure out a color. The game uses incomplete information, and with RIAL or DIAL the agents try to figure out the hidden message. (Foerster et al. 2016).

There have been successes with partial observability, but it requires more research. In a unified game-theoretic approach to multiagent reinforcement learning, it was explained that poker is one of the only main games in which success in partial observability was achieved (Lanctot et al. 2017). It was a big breakthrough, because it was unique, and many games could not have the same principles applied to

them. Games with incomplete information, while can be played and understood by humans, often require much more computation and different strategies than games with all actions available (Burch 2018). Thus, fewer games with incomplete information are tested than those with complete information.

I trained an AI on a simple incomplete action game – two player BS (game explained in methodology). Using Open AI Gym, I learned more about training games with incomplete information.

METHODOLOGY:

General Steps in the Method:

My methodology is based on a methodology similar to the one used in *Coin Run* - create a game environment, add variation into the game, test the game with an AI agent with an algorithm, and graph the result. I used the graphing code from a tutorial which I thought had great graphs and I added an extension seaborne to make them more visually aesthetic (Stable Baselines 3 Contributors, 2024). I referenced similar methodologies from other articles as well - learning what types of games were made and how AI was used and tested.*(Putten R 2018 , Train a Deep Q Network with TF-Agents).* The methodology also incorporates: *Deep reinforcement learning at the edge of the statistical precipice* in order to gather reliable data from the AI agent, and try to reduce bias by using ideas such as considering all the runs of a specific algorithm - not only successful runs (Agarwal et al. 2021).

Setting-Up the BS Game

In order to test a game with incomplete actions I coded a two player BS game in Python. Two player BS was chosen because it is simpler than games such as Poker or UNO, and can be understood and coded more easily.

I first made the game in Python, and made sure that the user could play against a scripted opponent. The game only uses part of a standard 52 card deck so that players don't know each other's cards. In each game, 30-45 possible cards can be used. A standard variation of BS was chosen. Players are dealt a certain amount of cards, and then they put down the cards in an order starting from ace, two, three, and so on. Players show what amount of cards they put down, and if someone calls "bs" then the cards that were put down are revealed. If the person putting down cards was truthful then the person calling "bs" takes the stack, and if the player putting down cards lied then they take the stack. The goal is to put down all the cards. (Tichvon, 2023).

The user types in the cards they want to put down in a certain format, and the scripted agent also plays. The scripted player was made to test the Python game itself, and to be used in the future to play with an AI agent. The scripted player says "bs" one out of four times (unless "bs" was called previously), and tries to always be truthful. If it can put the correct card in the correct order it will do so, otherwise it will put a card in the next possible order available. For example, if the scripted agent has to put down an

“ace” and it has no “ace” then it can put down a 2, 3, 4, etc.. The scripted agent is simple, and the goal of having it is to see if the AI can win against it.

The players start out with a random amount of cards (between 30 and 45 cards) so that the games have more randomness and so that the agent does not memorize putting down the same cards. Similarly to Coin Run, I want my agent to be able to generalize. Each player has between 15 to 20 cards and both players start out with the same amount of cards. The card deck is shuffled to a random order at the beginning of the game and then it is split between two players. The cards are then assigned randomly to the players. Each player gets a “hand”, and the cards they get are put into order.

```
order = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

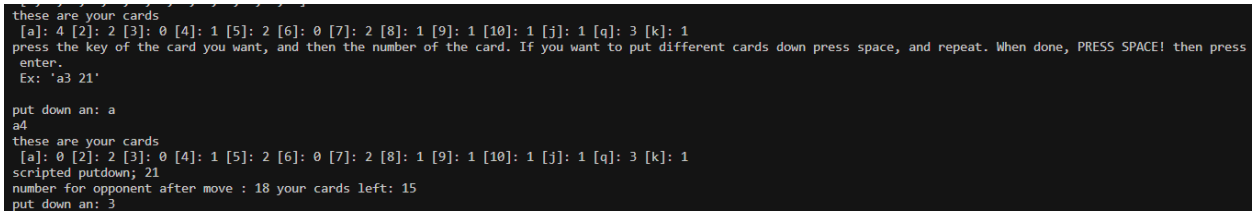
The code above shows an example order, in which 0 is king, 1 is ace, 11 is jack, and 12 is queen, and all the other cards are their numerical card values. As the card deck declines, the uncertainty of what cards the opponent has increases. With 30-45 cards, it can be guessed what cards the opponent has, but not with certainty.

Such as: [k]:1, [a]: 4, [2]: 2, [3]: 0, [4]: 1, [5]: 2, [6]: 0, [7]: 2, [8]: 1, [9]: 1, [10]: 1, [j]: 1, [q]: 3

(There is one king, four aces, two twos, zero threes, one fours, etc….)

I checked if certain parts of my game worked as I intended them to, by testing the game in certain cases. I made sure it would function as intended when the user, and when the scripted player said “bs” in

certain situations. Moves the scripted player took were recorded by printing them out for the user to see, and when it said a lie and “bs” was entered by the user, the cards went back to the scripted player.



Once the game itself was done, I put it into the Open AI Gym Framework - a framework for users to test agents on games they have made themselves (Brockman 2016).

Putting the Game into an Open AI Gym Framework:

The BS game was put into specific functions such as “step” (make the game go one timestep), “initialize” (set up the game environment) and “reset” (restart and set up a game when it is done). I use "episode" to describe a single game. It is assumed that the game is always played until a player wins. Therefore the episode generically consists of multiple timesteps. ‘Episode’ is an instance or length of a game, in which the amount of individual timesteps within vary. The Open AI framework is a commonly used framework for AI games, and my method uses it since it is openly available, useful and convenient for training an agent (Brockman et al. 2016).

For the framework to be used, an ‘observation space’ must be made for the AI agent - this is everything that the AI agent can see. With reinforcement learning, the agent does not know the rules of the game, it takes actions and considers their outcomes (Pettit J 2019). The observation space I intended

was: the cards the agent has, the amount of cards the agent has, the amount of cards the opponent has, the current card is supposed to place down - like an ‘ace’, and the amount of cards in the stack. I wanted the agent to consider this information when deciding what move to make, similarly to how when I play a game of bs I know the cards I have, the amount of cards I have, the amount my opponent has (although not as exact as the real number), the card the game is on, and the amount of cards in the stack.

The action space also had to be specified, and the action space was made to be (3, 13, 4, 3, 13, 4, 3, 13, 4, 3, 13, 4) to make any action available to the agent - even if the agent is not always able to take any possible action. The possible actions are ([“bs”, action, no action] (the three), [ace, two, three, four, five, six, seven,eight, nine 10, j, 1, k] (the 13), [1 card, 2 card, 3 card, 4 card]) (the 4) repeating 4 times for to create an action space with all actions available. The agent can say [0,7,4] which is “bs” and it does not put down a 7 and 4, while it can say [2,7,4] which says to take an action and put down a seven four times. I wanted the agent to be able to put down four individual cards, but still be able to say bs, and take any possible action. The action space is large, but it allows for all actions to be explored even if it might take more time training the AI

Choosing an Algorithm:

In order to train an agent an algorithm must be used. A common algorithm is proximal policy optimization (PPO). It was used because it worked in the framework and it allowed for the type of action space I specified. Not all algorithms allow certain shapes in the action spaces, such as “boxes” which I

used, there were discrete values in the action space. I imported it with Stable Baselines 3 and used it to train my agent.

Modify the Framework and the Imported Algorithm/Algorithms over time:

After the agent is tested, parts of the framework may be modified in order for the correct action space and observation space to be used. A big part of the method is debugging, and finding how certain changes impact how the agent learns. Parts of the code must be modified and corrected in order to find bugs that slow an agent down, or discrepancies in game logic. Also there should be certain reward considerations - different rewards can be given to help the agent learn certain behaviors.

Graph the Results of the Agents Wins - Once the Algorithm and Certain Parameters are specified:

I'm using graphs of reward over episodes (games the agent played) (how the wins of the agent change over time) and duration over episodes (how the lengths of the games change over time). The graph shows the individual outcomes of the games, but also the mean outcomes, to see how the mean changes.

RESULTS:

I graphed an agent's reward and game durations after training, modifying my code several times to account for different variations. The blue lines show the actual outcome of an episode, while the orange line shows the mean of the past 5 or 100 episodes (different graphs used different averages). For example,

even if the blue reward lines vary, the mean line can be used to see how the agent does over time on average.

Originally, I accidentally left out part of the scripted player, and the agent would just play itself - it would put down its own cards until the agent ran out of cards, which turned out to be around 8-10 moves according to Figure 1. I had certain variations in my code, which I trained on and graphed certain situations. I also realized that I accidentally showed the scripted player the opponents cards, and the actual cards in the deck. To me, this seems like more information given to the agent than I had intended, even though the numbers were excluded, so I decided to have results from two observation spaces, observation space 1 was accidental while observation space 2 was intended. An observation space is the values the agent sees, and the agent is only able to make decisions and load weights into the neural network from the observation space it sees.

Observation space 1: [cards of player 1, cards of player 2, cards in the stack, current card the game says to put down]

Observation space 2: [cards of player 1, number for player 1, number for player 2, current card the game says to put down, number of cards in the deck].

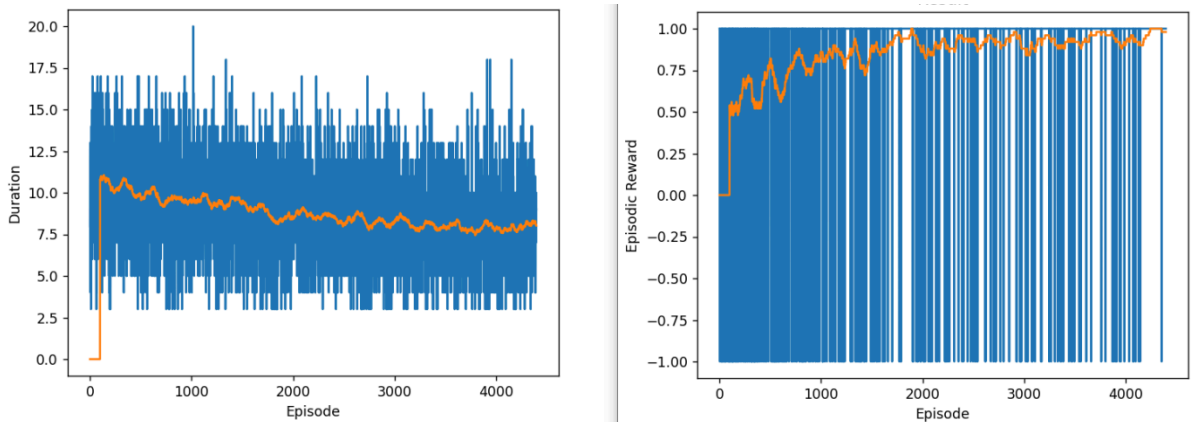


Figure 1: No Scripted Player

Caption 1:Using 3e5 Timesteps and graphing 100 episodes with the scripted player not fully included and a maximum of 500 timesteps in observation space 1.

I used PPO (Proximal Policy Optimization), a common algorithm that I could upload from Stable Baselines 3, an open source library with algorithms. I used it for 3e5 ($x*10^y$ form or $3*10^5$) timesteps to learn (it would learn for 300,000 timesteps), afterwards it would learn and graph results for 1000 episodes (full games that the agent would play). According to the graph the initial training took around 3000 episodes, and playing the game 1000 times showed that the agent on average got a mean of around 0.85 points based on the results from the final 1000 games in the end. The reward given for placing down all the cards was 1 point, and 0 points for not being able to place all the cards down.

For the second attempt, after realizing that the scripted player was not fully included, I made sure it could call “bs”, but I decided not to have it put cards down. The maximum length of time for a game was 500 steps for the second attempt (turns in an individual game), but I saw that the agent would play all

500. Then I set the max amount of steps to 1000, and even though the agent would play all 1000 timesteps I decided to keep it because 1000 is a very high number of turns for a bs game.

I also gave the agent rewards for calling “bs” correctly, and penalties for calling “bs” incorrectly: 0.05 points for correctly, and 0.05 points for incorrectly. The agent would receive a reward of 1 for putting all its cards down, and a reward of 0.5 for having more cards than the player who only called bs (which I later realized did not make sense, since the agent would have more cards than the scripted agent who only called “bs” because the scripted player could not put its half of the game deck down).

I debugged several scenarios in which the game logic did not function correctly. The average reward stayed between 8-10 points (the reward was inflated because of penalties and rewards for calling “bs” correctly or incorrectly), and the average episode length would always reach 1000 episodes.The code had a bug where if “bs” was called incorrectly the agent would still get a reward, thus inflating the rewards. The image below does not show an agent who learns, because the reward and duration stay almost constant over time.

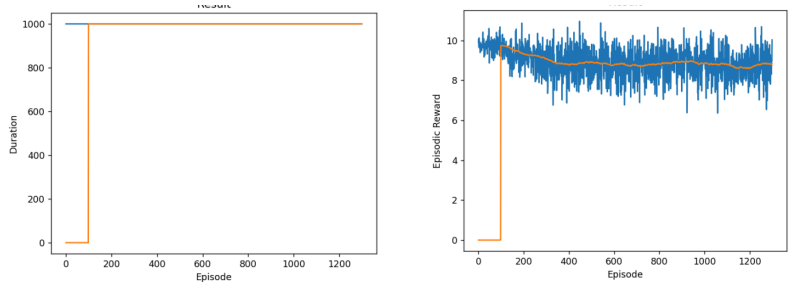


Image 1: An agent who does not learn

After I saw that “bs” was called incorrectly for the agent and I fixed that bug, I ran the code and got an agent who learned to lose:

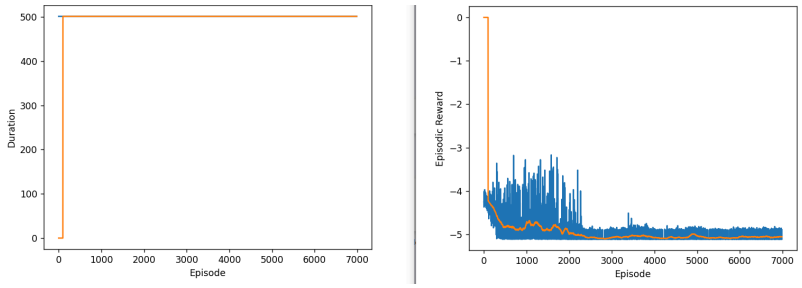


Image 2: An agent who learns to lose

The agent would not put down all its cards, and it got “bs” called on it or it called “bs” incorrectly many times. I saw that there was still something wrong with how the agent played “bs”. I found parts of the code that needed to be changed, such as how bs was called, and how the players cards were counted.

Figure 2 debugged parts of Figure 1 and it shows the agent learning over time and learning good behaviors leading to more rewards part of the time, and bad behaviors leading to less rewards as well.

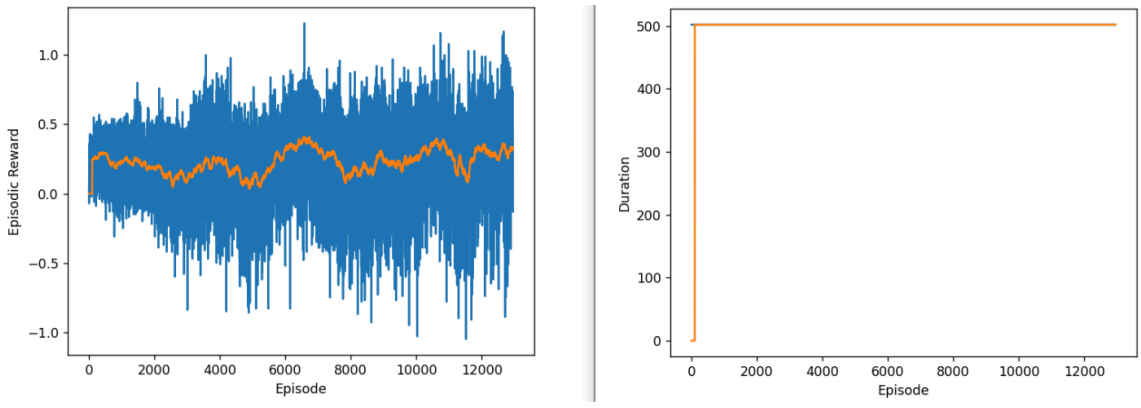


Figure 2: Adding the working Scripted Player

Caption 2:Using 3e6 Timesteps and graphing 1000 episodes with a scripted player only calling "bs" and a maximum of 1000 timesteps in a game in observation space 1

Figure 3 shows the code with one more bug fix - there was a line of code that would check if the agent could put down cards and it would say that if the amount of cards was less than the available amount the agent could put the cards down. It should have checked if it was less or equal to the number of cards so that the agent could reach 0 - before the agent was not able to do so. Figure 5 shows the change and it is also trained on games of maximum duration of 1000 timesteps instead of 500 steps as seen in Figure 4.

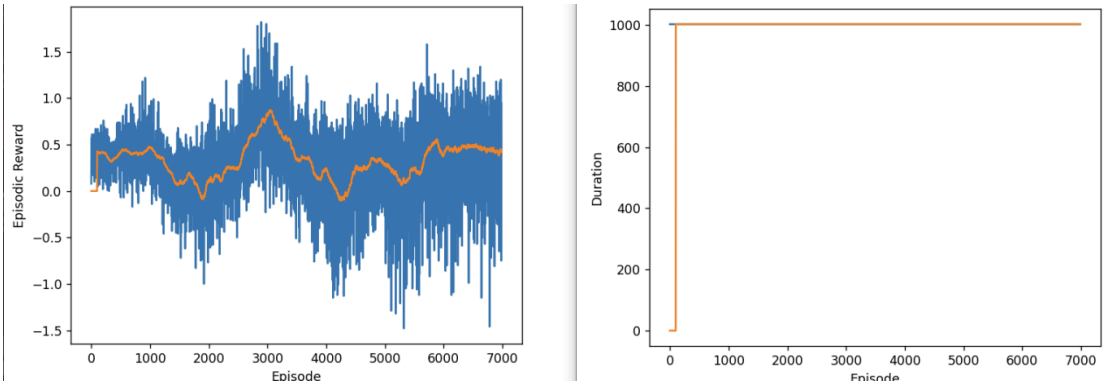


Figure 3: Allowing the Agent to Reach 0 Cards

Caption 3:Using 3e6 Timesteps in observation space 1 and graphing 1000 episodes with a scripted player only calling "bs" and a maximum of 1000 timesteps in a game with a code fix from Figure 2

I made sure the agent lost and gained cards correctly, "bs" was called correctly, and that the cards in the end were counted correctly as well. I got Figure 4, as the final result for playing against a scripted player who only said "bs". I trained the agent on 3e6 and 3e5 timesteps, and the first part of figure 4 shows how the game play normalized over time (it's hard to see what happened because the long time span shows changes less clearly) while the second part is more detailed and shows fewer episodes but more of how the duration of the game went down and how the episode result originally rose and then fell.

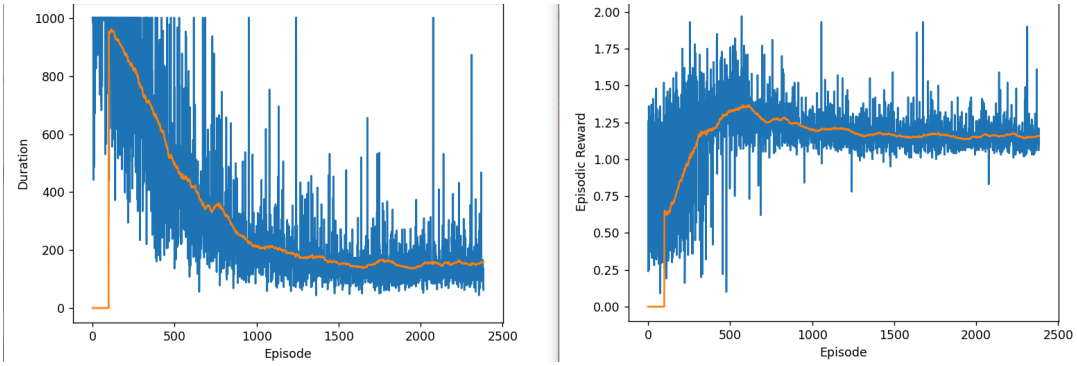


Figure 4: Debugged BS Game Pt.1

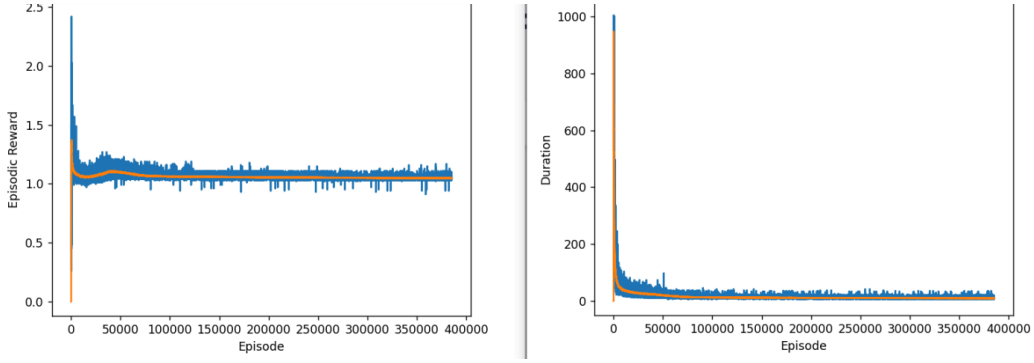


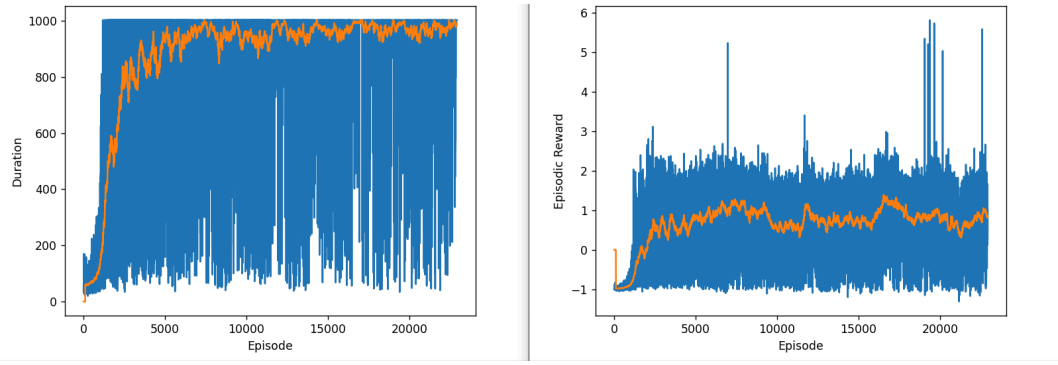
Figure 4: Debugged BS Game Pt.2

Caption 4:Using 3e6 and 3e7 Timesteps in observation space 1 and graphing 1000 episodes with a scripted player only calling "bs" and a maximum of 1000 timesteps in a game

While Figure 4 shows an agent that learns, it does not show an agent who would win against a scripted player. As I was training my agent, I realized that it is worth saving the weights for the learned neural network, because those can be used later, for example as a starting point.

For Figure 5 I combined weights I saved before, so after training the weights would be saved to a model. I used those weights to retrain the agent, playing against the real scripted agent (the one who puts down cards and says "bs"). I got an agent who did worse, but was more consistent.

Weights from this training the code in Figure 4 were used in Figure 5.



(Observation Space 1) Figure 5: Combined weights

Caption 5:Using 3e6 Timesteps in observation space 1 and graphing 1000 episodes with full a scripted player maximum of 1000 timesteps in a game

I modified how long the program would run, and the reward structure, and I found an interesting pattern. The agent would decrease the rewards it accumulated in games, but it would reach an equal amount of cards with the opponent. I saved images from training that would show that the agent would learn and decrease the rewards it received, while at the same time learning to put down cards and match with the opponent.

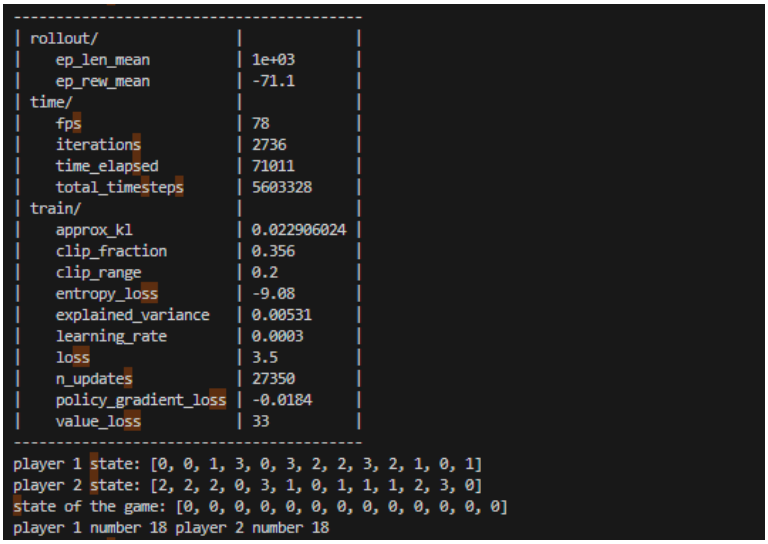


Image 3: Screenshots of some game statistics while running

Even though the agent learned to put down cards and win against a scripted agent who said “bs”, it did not win against a full scripted agent who would put down cards. Even though the rewards would decrease over time, the agent would tie by the amount of cards it had with the opponent.

For observation space 2, I rewrote my code to use number and list comparisons instead of string comparisons in order for it to function faster. I also modified the observation space to how I initially desired it to be - *the opponents cards are not able to be seen, and the agent can see the number of cards in the stack and not the stack itself.*

I ran my code with initial rewards for 1e6 episodes, to get Figure 6 for observation space 2. The agent

increased the duration, and reward over time, although the reward always stayed negative.

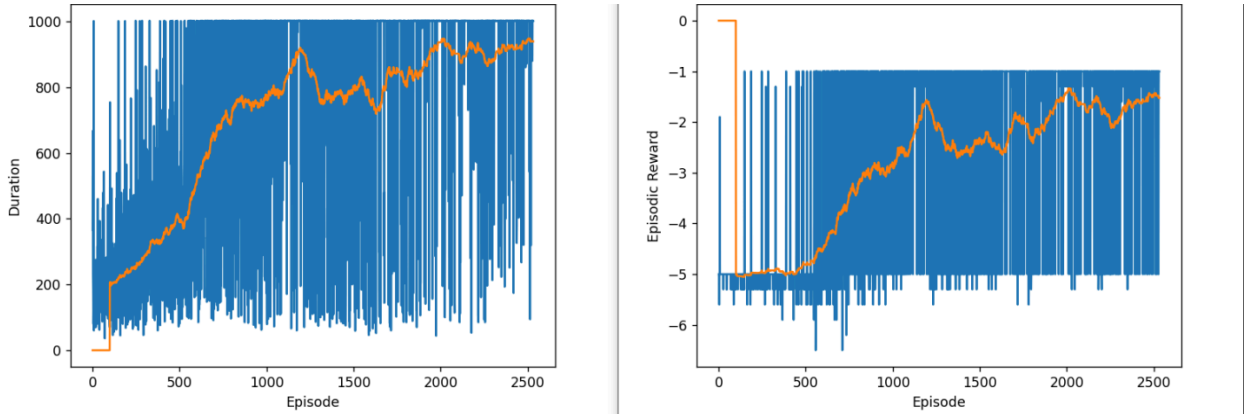


Figure 6 - Using Observation Space 2 with a full Scripted Player

Caption 6:Using 1e6 Timesteps in observation space 2 and graphing a full scripted player

For figure 7, I saved the weights from figure 6, changed the reward function to not reward "bs" for 1e6 timesteps. The episodic reward decreased over time, and the duration increased over time. The agent would still put down cards, but it would do it less frequently.

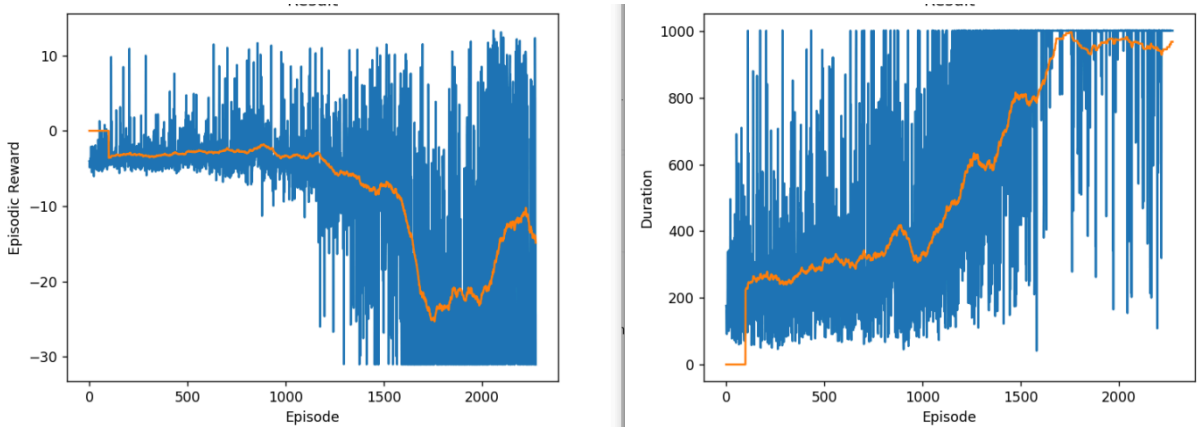


Figure 7: Training with Figure 6 Weights

Caption 7:Using 1e6 Timesteps in observation space 2 with a full scripted player with weights from Figure 6 and a maximum of 1000 timesteps in a game.

```
number for player 1: 29
number for player 2: 0
steps done: 847
player 1 state: [1, 2, 2, 3, 2, 1, 2, 3, 4, 2, 3, 1, 3]
player 2 state: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
state of the game: [0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 1, 0]
player 1 number 29 player 2 number 0
player cards [11, 5, 2, 0, 11, 7, 4, 11, 1, 9, 11, 6, 5, 4, 6, 3]
player cards [9, 8, 11, 6, 4, 10, 3, 6, 5, 9, 4, 5, 4, 3, 12, 2, 3, 6, 12, 8, 11, 2, 7, 0, 2, 2, 11, 10, 1, 1, 5, 11]
player 1 state: [1, 1, 1, 1, 2, 2, 2, 1, 0, 1, 0, 4, 0]
player 2 state: [0, 1, 3, 2, 1, 1, 1, 0, 2, 1, 2, 0, 2]
state of the game: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
player 1 number 16 player 2 number 16
player cards [5, 3, 1, 8, 6, 0, 2, 9, 11, 2, 3, 9, 6, 1, 2, 8, 5, 7, 10, 5, 7, 0]
player cards [11, 3, 5, 0, 1, 9, 5, 10, 1, 9, 5, 3, 11, 8, 2, 9, 2, 1, 10, 0, 7, 12, 11, 0, 8, 6, 7, 6, 1, 3, 6, 0, 6, 4, 12, 4, 10, 2, 3, 7, 9, 2, 4, 8]
number for player 1: 40
number for player 2: 3
steps done: 1001
player 1 state: [4, 4, 3, 4, 3, 3, 4, 3, 2, 4, 3, 1, 2]
player 2 state: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0]
state of the game: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
player 1 number 40 player 2 number 3
player cards [12, 4, 12, 3, 1, 1, 8, 5, 8, 6, 8, 5, 11, 7, 4, 11, 10]
player cards [10, 9, 8, 4, 12, 11, 6, 7, 7, 2, 3, 8, 12, 5, 11, 11, 5, 0, 12, 1, 2, 8, 7, 4, 5, 4, 1, 3, 3, 6, 10, 1, 4, 6]
number for player 1: 29
number for player 2: 0
steps done: 601
```

Image 4: Image of some game statistics while running

For Figure 8 I added the weights from figure 7 (which had added weights from figure 6), and I trained the result on 1e7 and 2e6 timesteps, because I found it interesting how on part one (1e7) the rewards increased and then rapidly decreased, so I wanted to find out if it was something random and zoom in.

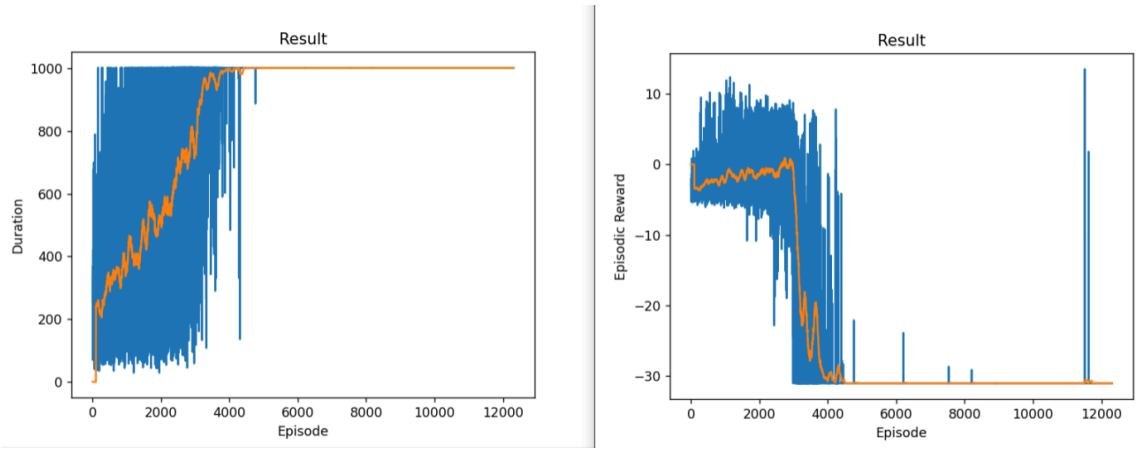


Figure 8 Pt. 1

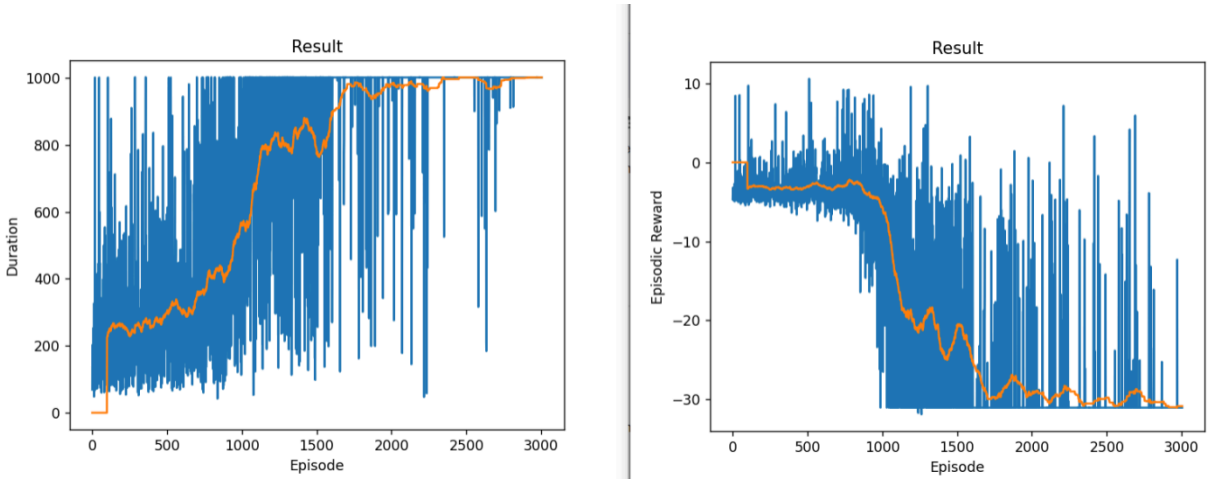


Figure 8 Pt. 2

Caption 8:Using 1e7 and 1e6 Timesteps in observation space 2 using weights from Figure 7 to train an agent with a maximum of 1000 timesteps per game.

```
player 1 number 18 player 2 number 18
player cards [4, 2, 1, 0, 1, 11, 6, 0, 5, 3, 7, 6, 9, 10, 9, 1, 11]
player cards [8, 0, 2, 6, 3, 0, 7, 5, 6, 7, 11, 2, 10, 6, 8, 8, 9, 12, 12, 7, 9, 1, 0, 4, 6, 5, 9, 1, 4, 1, 10, 2, 3, 11]
player 1 state: [2, 3, 1, 1, 1, 1, 2, 1, 0, 2, 1, 2, 0]
player 2 state: [1, 0, 2, 1, 1, 1, 2, 2, 3, 1, 1, 0, 2]
state of the game: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
player 1 number 17 player 2 number 17
player cards [12, 10, 11, 7, 12, 3, 4, 12, 4, 6, 10, 5, 1, 5, 8]
player cards [6, 8, 0, 12, 2, 3, 4, 10, 2, 1, 7, 12, 12, 1, 11, 1, 9, 8, 4, 11, 5, 0, 3, 7, 5, 3, 10, 6, 1, 5]
----------------------------------------
| rollout/                |             |
|    ep_len_mean          | 999         |
|    ep_rew_mean          | -30.8       |
| time/                   |             |
|    fps                  | 54          |
|    iterations           | 971         |
|    time_elapsed         | 36512       |
|    total_timesteps      | 1988608     |
| train/                  |             |
|    approx_kl            | 0.036153346 |
|    clip_fraction        | 0.397       |
|    clip_range           | 0.2         |
|    entropy_loss         | -13.3       |
|    explained_variance   | -0.00975    |
|    learning_rate        | 0.0003      |
|    loss                 | 0.667       |
|    n_updates            | 9700        |
|    policy_gradient_loss | -0.0201     |
|    value_loss           | 6.85        |
----------------------------------------
```

Image 5: Image of agame statistics while running

The agent would often put down the same amount of cards as it initially had, which I found surprising that it chose this as the logical move in both observation spaces.

The AI agent was able to learn from a game of BS. In observation space 1 the AI agent was able to improve and learn over time with the agent who only called “bs” in observation space 1, it learned to tie with the scripted player as well. The figure 4 in observation space 1 is the only figure that shows an AI

agent trying to maximize the rewards and lowering timesteps in successive episodes. The agent also learned to do certain behaviors as seen by rewards rising and falling over significant periods of time. In observation space 2, the agent also learned to put down cards, and it learned to tie with the scripted player as well. It's important to see how AI learns with incomplete information, in order to learn what information is really needed to give AI and how AI can use incomplete information to play games and do tasks.

DISCUSSION AND ANALYSIS:

The agent is able to adapt to changing code, and learn certain behaviors, improve and put down cards. This is significant, because it shows that the game itself is just the problem the user wants to be solved, but the agent can learn to solve another problem if the code actually addresses a different task. For example, if the code does not function as intended, the agent learns and tries to succeed at the unintended task. Also, putting down all the cards is a challenge for the AI agent, because the way my reward is structured, having 16 cards vs. 6 cards in the end leads to the same reward. While the reward of having put all cards down is significantly more, it might be difficult for the agent to realize that, and it was not incentivized enough to find that reward. The AI does not have the same goals as the user, and maximizing rewards does not always mean learning the best behaviors. Also, I found it interesting how different weights of a model could be combined - I did not think of it initially, but then I realized that saving past

experiences would be useful to change the reward structure and see how an already trained agent would adapt to that change.

As I kept modifying the code and finding certain parts that I had not considered before, such as making sure that the agent could not call bs right after the scripted player called “bs”. If there are 0 cards in the stack (since in BS someone has to take the cards after “bs” is called) and the agent has to put a card down, then there is no reason for “bs” to be considered. I made it so that the agent and scripted player disregarded actions with “bs" if “bs” had been called in the previous turn. While there is a small chance the agent would use “bs” as its 4 actions (shown in method), the chances of that were unlikely, and if the agent did so then it would be detrimental since the agent would not be able to get rid of any of its cards. Changing the reward structure changed what the agent would focus on, such as giving a bigger reward for calling bs or losing bs could change how the agent called bs and how it tried to avoid it being called on the agent. By changing parts of the code and considering certain limitations of how the code worked at the moment, the results the agent gave varied significantly. Just because the game with the user (when a human could type the cards in) vs. the scripted player worked correctly, did not mean that as the environment was reset and the agent and scripted players played each other automatically there would not be other difficulties. With further modifications, Figure 4 showed an agent who appeared to be learning against just an agent who could say “bs”. The rewards would increase and decrease over certain time intervals, although the durations would stay at 500 or 1000 because the code would not let the agent put

all its cards down . Figure 5 was similar to Figure 4, but Figure 5 had more variety in how the agent learned, and it included an important fix - making sure the agent could reach 0 cards. While this fix seems influential, the difference between Figure 5 and Figure 4 is not apparent. After getting results from figure 5 I double checked if the cards were counted correctly in the end, and I found that they were not. Figure 6 includes the fix of having the cards counted correctly. Part of the reason the past figures showed the agent reaching the max amount of timesteps was because the agent's cards did not add to 0 when they should have, leading it to be impossible to end the game without reaching the full amount of timesteps (either 500 or 1000 depending on the figure). Figure 6 has two parts, one with the agent training 3e6 timesteps, and 3e5 timesteps. The 3e6 graph shows the agent stabilizing the reward and duration of the game it plays, but the figure also shows too long of a time interval. Then I trained the agent on 3e5 timesteps, and the graph shows how rewards increase and the duration of the game decreases, with some of the rewards decreasing as the durations of the games reach a lower number. Figure 6 shows the agent learning, because over time the agent adapts and tries to get more rewards while decreasing the length of the games.

I am not aware of another BS game that someone else has made and trained an AI with that I could compare my results to, which poses as a limitation to comparing if Figure 4 (observation space 1) and Figure 7 and 8 (observation space 2) shows an agent who plays the game well, or just an agent that learned to improve its initial and put down cards. While I don't know how well the agent actually played,

the results do show that the agent's performance got better over time and that it learned positive actions in Figure 4, and it was able to keep a constant amount of cards in Figure 6. I decided to keep Figures from code that would later be modified to show how AI learns on tasks regardless of logic issues in the game. The different observation spaces also show how an agent is able to adapt to seeing different parts of the game, and how that changes how the agent plays and learns over time.

CONCLUSION AND FUTURE DIRECTION:

Games such as BS are important because they show incomplete actions, there are not a lot of games with partial observability that have been trained before. There are people working to solve problems with incomplete actions, or learning with multiple agents to solve complex tasks or games(Canese Lorenzo et al 2021, Gmytrasiewicz, P. J., & Doshi, P. 2005). While two player BS is not a complex game, it provides insight to how an agent can use partial observability and learn even when not all actions are visible. I hope that my bs game can help others use incomplete actions and train an agent to learn as well. I used approaches such as adding weights, changing reward network weights, and described how I coded, tested, and chose my game. I hope those approaches can be useful to someone else wanting to train an incomplete information game. While Poker was a breakthrough, its principles were not necessarily able to be directly applied to other games, and similarly, parts of BS might not be applicable to other situations. Since each game is unique, BS might be an outlier. Also the results do not

show that the agent learns well, rather that the agent learns to put down cards, and can learn to match and put down the same amount of cards as the scripted player.

The results show that an agent can learn and improve on a game with incomplete actions or observability. Even though observation space 1 offered a lot of information, it did not provide how many cards the agent itself, the game, or the opponent had, leading to some lack of information as well. Observation space 2 does not show the individual cards the opponent has, or the individual cards in the stack, leading the agent to learn from more missing information than observation space one. The results show that an agent can even improve if the code is not functioning as desired, and how tests over time and improvements over time can lead to different outcomes and graphs. Future work would include changing the reward function more to see if the agent learns better when given other rewards, visualizing the game to see what the agent does step by step (to make a model and run through the agent's logic). I would like to try combining different weights, and reward structures, to make an agent who puts down all its cards against the scripted agent and one who plays “bs"” efficiently. It would be desirable to develop mathematical models of how a BS game should be optimally solved. This would allow comparing results of a specific learning algorithm to what is known from theoretical analysis of the game. Also, it would be good to develop models of how different playing styles will be affected by an agent's observation space. Also, different algorithms can be compared and researched similarly to how (Samsuden et. al 2019) used

and graphed different algorithms to see how well they performed. Overall, I'm really happy to have made a BS game that I can use an AI agent to learn and improve in.

SIGNIFICANT ACHIEVEMENT:

When I saw that the agent would have the same amount of cards as the scripted player, I thought that my code had a mistake. Maybe it was printing out the initial start states? I'm still not fully sure why the agent was matching the amount of cards it had, but it is logical - having 21 cards vs. 21 cards gets the same end reward as 1 card vs. 40, so maybe it learned to just match the amount of cards the opponent has (maybe that was easiest). Through my approach I was able to see interesting situations in which the agent learned logical behaviors that I had not previously considered.

ACKNOWLEDGEMENTS:

Thank you Dr. Soto for reading my paper and working with me on my research project in research class – your insights and suggestions are really valuable!

References

Agarwal, R., Schwarzer, M., Castro, P. S., Courville, A. C., & Bellemare, M. (2021). Deep reinforcement learning at the edge of the statistical precipice. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, *34*, 29304-29320.

Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems." *IEEE transactions on systems, man, and cybernetics* 5 (1983): 834-846.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Burch, N. (2018). Time and space: Why imperfect information games are hard.

Canese, Lorenzo, et al. "Multi-agent reinforcement learning: A review of challenges and applications." *Applied Sciences* 11.11 (2021): 4948.

Cobbe, K., Klimov, O., Hesse, C., Kim, T., & Schulman, J. (2019, May). Quantifying generalization in reinforcement learning. INTERNATIONAL CONFERENCE ON MACHINE LEARNING (pp. 1282-1289). PMLR.

Dayan, P., & Hinton, G. E. (1992). Feudal reinforcement learning. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, *5*.

Foerster, Jakob, et al.METHOD "Learning to communicate with deep multi-agent reinforcement learning." *Advances in neural information processing systems* 29 (2016).

Haridy, R. (2017, December 27). *2017: The year AI beat us at all our own games*. New Atlas. Retrieved April 7, 2024, from https://newatlas.com/ai-2017-beating-humans-games/52741/

Knight, W. (2017, January 23). *Why Poker Is a Big Deal for Artificial Intelligence*. MIT Technology Review. Retrieved December 10, 2023, from https://www.technologyreview.com/2017/01/23/154433/why-poker-is-a-big-deal-for-artificial-intelligence/

Lanctot, M., Zambaldi, V., Gruslys, A., Lazaridou, A., Tuyls, K., Pérolat, J., ... & Graepel, T. (2017). A unified game-theoretic approach to multiagent reinforcement learning. *Advances in neural information processing systems*, *30*.

Nichol, A., Pfau, V., Hesse, C., Klimov, O., & Schulman, J. (2018) SOME METHOD. Gotta learn fast: A new benchmark for generalization in rl. *arXiv preprint arXiv:1804.03720*.

Pettit, J. (2019, August 6). *Introducing gym-snake-rl*. Jacob Pettit. SOME METHODRetrieved November 1, 2023, from https://jfpettit.svbtle.com/introducing-gym-snake-rl.

Putten, R. v. (2018, March 13). *FrozenLake and Dynamic Programming*. LinkedIn. Retrieved November 13, 2023, from https://www.linkedin.com/pulse/frozenlake-dynamic-programming-rob-van-putten.

Ravichandiran, S. (2018). *Hands-on reinforcement learning with Python: master reinforcement and deep reinforcement learning using OpenAI gym and tensorFlow*. Packt Publishing Ltd.

Samsuden, M. A., Diah, N. M., & Rahman, N. A. (2019, October). A review paper on implementing reinforcement learning techniques in optimizing games performance. In *2019 IEEE 9th international conference on system engineering and technology (ICSET)* (pp. 258-263). IEEE.

Tichvon, C. (2023, 2). *How to Play B.S. - Card Game Rules & Instructions*. The Game Farm. Retrieved February 21, 2024, from https://the-game-farm.com/card-games/bs/

Train a Deep Q Network with TF-Agents. (2023, February 16). TensorFlow. Retrieved November 1, 2023, from https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial.

Stable Baselines 3 Contributors. (2024, April 1). *Stable Baselines3 Documentation*. Stable Baselines3 Documentation. Retrieved April 10, 2024, from https://stable-baselines3.readthedocs.io/_/downloads/en/master/pdf/