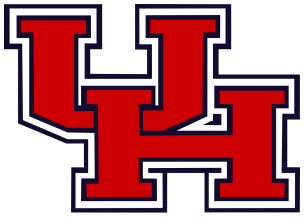
Artificial Intelligence

Final Project Report

Submitted By

Md Mahin (PSID: 1900421)
Amutheezan Sivagnanam (PSID: 1894948)
Shanto Roy (PSID: 1894941)

Department of Computer Science
University of Houston



July, 2020

Contents

List of Figures

Optimizing Experience Replay for Partially Observable Environment: Case Study on No Limit Texas Hold'em Poker using DQN

Md Mahin
University of Houston
Houston, Texas
mdmahin3@gmail.com

Amutheezan Sivagnanam
University of Houston
Houston, Texas
asivagnanam@uh.edu

Shanto Roy
University of Houston
Houston, Texas
sroy10@uh.edu

ABSTRACT

Experience replay is used to mitigate problems such as: correlation among samples, and drastic policy change during nonlinear function approximation of Q-function used in methods like Deep Q-Network (DQN). The learning process of any deep learning based agent can further be improved for any highly stochastic, partially observable multiple round games by incorporating domain specific information in designing and utilizing the replay buffer for experience replay. In this work, we address such issues for No-limit Texas Holdem Poker game. While saving observations within replay buffer for training procedure, no-limit holdem poker may face several issues: first, significant variation in number of samples within the replay buffer from each round; second, high number of similar information states within the experience buffer from each round; and third, highly varying stochasticity from each round. Therefore, in this work, we consider all three issues as independent problems, and developed three different solutions separately as three agents. Out of our three agents two agents shows poor reward distribution curve against a random agent in comparison to the base agent. But all three agents have winning percentage of 78.1%, 66.1% and 67.3% respectively against the base agent for 1000 games.

KEYWORDS

Experience Replay, reinforcement Learning, DQN, Poker Game, Partially Observable Games

1 INTRODUCTION

Experience replay is used to increase sample efficiency and to stabilize the training procedure of non-linear reinforcement learning like DQN [3, 5]. The nonlinear function approximator when used with temporal difference learning can make the learning procedure unstable and can also lead to divergence of Q-function [5]. There are several reasons for that: first, correlation among the sequence of observations, then, second, drastic change of policy due to small updates, and third, the correlation between action values and target values in form of td-error [5].

Experience replay is used to solve the first two problems by initially storing the sample before training and then using mini batches of randomly chosen samples from previously stored samples for training. The third problem can be solved by introducing a second neural network for training, that is a copy of the network used to predict action-values, but freezes before a certain interval till it is updated again [5]. Experience replay ensures a sample to be used multiple times for the training procedure. One benefit of experience replay is that it reduces the required experience. As a result it can reduce the number of interactions the agent makes with the environment, making the training procedure cheaper [3]. The design process of replay memory enables us with two design choices: first, what to save in the memory; and second, which experience to replay [3].

However, game like no-limit Texas Holdem poker enables us with more opportunity to improve learning with experience replay. While saving observations within replay buffer for training procedure, due to the imperfect information from the environment, no-limit holdem poker may face several issues. First, significant variation in number of samples within the replay buffer from each round. It could become a critical issue as the observibility of environment from each round is different, probably overfitting the agent for specific sample and underfitting for other samples. Second, high number of similar information states within the experience buffer from each round. They could become a problem as they train the agent only with same pattern of information. Third, highly varying stochasticity from each round. This is a problem as same observations can give completely different reward and to learn all the patterns we need to spend some huge amount of time in training. Therefore, we seek answers to the following two major questions:

Q1. How to formulate different *solutions* to train a dqn agent better for the partially observable *No Limit Texas Holdem Poker* environment that provides varying incomplete information in each round?
Q2. What are the experiment results and evaluation for different *algorithms*?

We answer the first question by forming three solutions to three different issues for a partial observable game like no-limit Texas Holdem poker (Section 3). Here, in this work, we consider all three above-mentioned issues as independent problems, and seek solutions separately. To address the first problem, our proposed method divides the replay buffer into four sub-parts matching the game rounds, and utilizes equal samples from each sub-buffer for training. To address the second problem, our method represents a whole information state with only three observations. Finally, to address the third problem, we propose inserting synthetic samples within the replay buffer based on the stochasticity of each round.

We answer the second question by evaluating our proposed methods in terms of training our agent and let it play with a regular DQN-based agent (Section 4). Here, we arrange a tournament of 500 games with each of our trained agents against the trained base agent. From the reward distribution, our first method discovers more wider reward space than the base agent, and the last two agent's reward space curve is always below the base agent. However, in each tournament each of our agents significantly outperformed the base agent, implying our agents are better trained. Based on the evaluation we find, the agent trained using method 1 performed best, then using method 2, and method 3 respectively.

Scope: This work only considers strategy and evaluation for no-limit holdem poker game. While evaluating the performance, we consider the *rlcard DQN agent*[1] as the opponent of the agent trained based on our proposed algorithms.

Organization: The rest of the paper is organized as follows: Section 2 presents necessary background details and definitions of our work. Then, Section 3 discuss and elaborate our proposed three methods and associated algorithms of optimization and filtering of the experience replay buffer. Finally, we provide the evaluation of our algorithms in Section 4 followed by related works in Section 5 and concluding remarks.

2 BACKGROUND

2.1 Stochastic Games

Stochastic games are the dynamic interaction model where the environment changes based on the behaviour of the players. In stochastic games the transition probability from state to state is jointly controlled by the players participating in the game [4].

Partially Observable Environment: Partially observable environment are the environments when agents cannot see the full state of the environment before taking any action. If full state of the environment is visible to an agent than it is fully observable environment. When decision makers have differing information about the states of the game we know that as imperfect information game [8].

Information states: Let N is the finite set of players. H is the finite set of histories of actions. $Z \subseteq H$ be the terminal sets. $A(h) = \{a : (a, h) \in H\}$ be the actions available after a non-terminal history $h \in H$. Let P is a player function and $P(h)$ is the player who takes an action after history h. Now, for each player $i \in N$ a partition $I_i = \{h \in H : P(h) = i\}$, with the property that $A(h) = A(h')$, whenever h and h' are in the same member of the partition. Now for $i_i \in I_i$ we denote by $A(i_i)$ the set $A(h)$ and by $P(i_i)$, the player $P(h)$ for any $h \in i_i$. I_i is the information partition of player i, $i_i \in I_i$ is an information set of player i [8].

Zero Sum Extensive Game: Lets $N = \{1, 2\}$. For each player $i \in N$, if u_i is the utility function from the terminal states Z, and $u_1 = -u_2$ it is called zero sum extensive games.r[8].

[1] http://rlcard.org/rlcard.agents.html#module-rlcard.agents.dqn_agent

2.2 Reinforcement Learning (RL)

RL is considered as a part of machine learning where a software agent perceive data from environment states, and takes actions while maximizing the combining reward for each state-action pair. RL can be modeled as a *Markov decision process* except where we do not have any transition and reward model. Therefore, the major components of RL are: agents, set of environment states S, set of actions A, and set of rewards R for associated actions.

Now, if there is a state $s \in S$, an action $a \in A$, a reward $r \in R$, discount factor γ, where $0 \leq \gamma \leq 1$, and horizon or time step h, the final goal is to find an optimal policy π^* such that,

$$\pi^* = argmax_\pi \sum_{t=0}^{h} \gamma^t E_\pi[r_t]$$

Here, a policy is defined based on what action an agent choose at each step after observing a particular state. It is sort of mapping from steps to actions, i.e., $\pi(s_t) = a_t$.

2.3 Temporal Difference Learning

Temporal Difference (TD) learning is an area of Reinforcement learning that is *model-free*. A model-free learning does not require explicit transition or reward model, that are associated with the traditional Markov Decision process. Rather, it learns by bootstrapping (random sampling based on bias, variance, confidence intervals, or prediction error) from the value function (e.g., Q-learning).

Q-learning: Q-learning is a model-free reinforcement learning algorithm that can learn a policy in a highly stochastic game with stochastic transitions and rewards. Using this algorithm, we can approximate the optimal action-value function of a stochastic game like no-limit Texas Holdem Poker. Here, we obtain a policy from the Q-function and then evaluate it using TD methods to obtain the next Q-function.

Deep Q-networks: Deep Q-network uses high dimensional sensory inputs for a deep neural network and learns policies accordingly. The neural network actually outputs a vector containing all the Q-values for corresponding actions. First, the DQN stores samples in a experience replay buffer with existing policy and fetch random samples from there. Then it uses the samples to update the Q network.

Experience Replays: Experience replay is used to reuse important experiences from the past. However, the samples are chosen randomly rather than selecting the sequential experience results. Random samples works well for stochastic games as sequential experiences are highly correlated and worsens the performance while playing games over time.

TD Error: TD error defines the deviation of the current prediction function from the condition for current input. In another word, it differentiates the estimated reward at any given state from the original reward received.

3 METHODOLOGY

3.1 Problem Formulation

No Limit Texas Holdem Poker: *No Limit* Texas Holdem Poker is a partially observable imperfect information game. It is also a zero sum game. In each complete game, there are four round. The observability of the game environment changes this way:

- *First Round:* A player can only see his two cards out of nine cards.
- *Second Round:* A player can see his two cards plus additional three cards from the public cards out of nine cards.
- *Third Round:* A player can see his two cards plus four public cards out of nine cards.
- *Fourth Round:* A player can see his two cards plus all five public cards out of nine cards.

Therefore, we see, the observability of the environment at each round is not the same.

On the other hand, in every round a number of bettings takes place between the opponents. The number of bets can be zero to any number k. During this betting time we see almost similar state patterns, with only little difference in the amount of chips in the pot. the number of bets in that state might provide similar and not very informative states to the agent. We define these similar states as *information sets*.

Another characteristics of this environment is the stochasticity of the game that also differs round to round. For example, at the first round of the game, when most of the cards are hidden, the highest pattern can have two aces from any two decs. However this pattern can result in different rewards in different games. On the other hand as the environment becomes more visible at different rounds, the stochasticity of each round decreases.

Experience Replay: While using normal experience replay in these kinds of situations a number of problems can occur.

(1) As the number of bets in each round can be different, the number of the observations in the experience buffer before can be significantly different. That might result in some rounds contributing very little in the training process, resulting in less learning for that round.
(2) The observations from each round that are being generated due to the betting are very similar. Intuitively we can say not all these samples are useful for efficient learning of our model.
(3) The high stochasticity of each environment can mislead. For example different rewards from each observation can deceive the agent to take some action. The problem can be solved by enough training, but due to high stochasticity the training period might need to be increased significantly.

Therefore, in this work, we address all these problems independently. We propose three methods that separately address all these problems, and the reasoning behind them.

3.2 Tuning Experience Replay

Method 01- Dividing the replay buffer: This method is created with the intuition to use same number of observations from each round to train the agent, so that no individual pattern of observations does not overfit the agent. We divide our replay buffer into *four* different parts of the same size. We have ensured our training procedure is invoked when all sub-buffer have the same number of samples. However, this way, we are sacrificing observations from some rounds till other sub-buffers are not full and that might be an issue.

During our training procedure we randomly draw an equal number of samples from each sub-buffers for the training. Algorithm 1 presents the algorithm of our first method. Here, parameters tt is the buffer threshold i.e. we wait till each of our buffers are filled. n is the number of trainable samples that is drawn from each buffer for the training. The "`ignore()`" function will ignore an observation without adding it to the buffer, "`add()`" function adds an observation to the respective sub buffer, and "`draw_random(n)`" function draws n number of samples from each respective sub buffer.

Algorithm 1: Replay Buffer Division

```
/* Used Variables:                                                    */
/* sub_buffer = Subset of Primary buffer, tt = buffer
   threshold, n = trainable samples                                   */
/* Used Functions:                                                    */
/* ignore() = ignores samples, add() = add samples to buffer,
   Train_model() = trains agent                                       */
foreach round do
    i = 1
    if sub_buffer[i] == tt then
        ignore(observation)
    else
        sub_buffer[i].add(observation)
if sub_buffer[1] and sub_buffer[2] and sub_buffer[3] and
  sub_buffer[4] == tt then
    S_1 = sub_buffer[1].draw_random(n)
    S_2 = sub_buffer[2].draw_random(n)
    S_3 = sub_buffer[3].draw_random(n)
    S_4 = sub_buffer[4].draw_random(n)
    Train_model ( S_1, S_2, S_3, S_4)
```

Method 02- Filtering samples from each information state: Our second method is designed to control similar states or information sets within the replay buffer. It filters out the redundant observations from each information state within our replay buffer. For this method we have used the same replay buffer structure from the base dqn in the RLCard. However, to store only the meaningful samples within the replay buffer, we represent the whole information state with only three observations:

i. *min_observation:* The observation that have minimum TD error with its previous state within the same information state. We know TD error calculates the difference between

expected reward and obtained reward. The reason for retaining the sample with minimum TD error is, it is closest to the optimal reward.

ii. *max_observation:* The observation that have maximum TD error with its previous state within the same information state. The reason for retaining this sample is, it makes the most significant change to the current state.

iii. *mid_observation:* The observation that have maximum TD error with its previous state within the same information state. The reason for retaining this sample is, it makes the most significant change to the current state.

Algorithm 2 presents the algorithm of our second method.

Algorithm 2: Filtering Samples from each Information State

```
/* min(TDE_{S_{i-1}}) = observation having minimum TD error from
   previous state, max(TDE_{S_{i-1}}) = observation having maximum
   TD error from previous state, min(R_{S_{i-1}}) = minimum
   observation reward, max(R_{S_{i-1}}) = maximum observation
   reward                                                    */
foreach round do
    /* Calculate min, max, and mid observations             */
    min_observation = min(TDE_{S_{i-1}})
    max_observation = max(TDE_{S_{i-1}})
    synthetic_pot_size = (min(TDE_{S_{i-1}}) + max(TDE_{S_{i-1}}))/2
    synthetic_reward = (min(R_{S_{i-1}}) + max(R_{S_{i-1}}))/2
    synthetic_state = create-state with synthetic_pot_size ,
      observable cards and action
    mid_observation = create_obersvation with
      synthetic_state and synthetic_reward
    /* Add observations to memory buffer                    */
    memory_buffer.add(min_observation)
    memory_buffer.add(max_observation)
    memory_buffer.add(mid_observation)
```

Method 03- Addressing Stochasticity by Bootstrapping Samples. Purpose of this method is to utilize the varying stochasticity of each round for the training procedure. This method retains the steps from our Algorithm 2. The only difference is, here to address *stochasticity* of the different information states, we introduce different amounts of synthetic samples within each round. The synthetic samples are exactly the same three samples from the *Method 02* with inverted reward or reward multiplied by −1. We have decided to include synthetic samples based on the extent of the stochasticity of each round. For example, as the round one is highly stochastic, we decide that 50% time of our game we will add any synthetic sample, as in the first round due to most imperfect information most of the states can give completely different reward. For round two it is 25%; for round three, it is 20% and for round four it is 15%. We have decreased the amount of synthetic samples in each round as the visibility of rounds increases and thus stochasticity of the rounds decreases. Here, the numbers are chosen randomly. Algorithm 3 presents the algorithm of our *Method 03.*

Algorithm 3: Bootstrapping Samples

```
/* min(TDE_{S_{i-1}}) = observation having minimum TD error from
   previous state, max(TDE_{S_{i-1}}) = observation having maximum
   TD error from previous state, min(R_{S_{i-1}}) = minimum
   observation reward, max(R_{S_{i-1}}) = maximum observation
   reward                                                    */
foreach round do
    /* Calculate min, max, and mid observations             */
    min_observation = min(TDE_{S_{i-1}})
    max_observation = max(TDE_{S_{i-1}})
    synthetic_pot_size = (min(TDE_{S_{i-1}}) + max(TDE_{S_{i-1}}))/2
    synthetic_reward = (min(R_{S_{i-1}}) + max(R_{S_{i-1}}))/2
    synthetic_state = create-state with synthetic_pot_size ,
      observable cards and action
    mid_observation = create_obersvation with
      synthetic_state and synthetic_reward
    /* Add observations to memory buffer                    */
    memory_buffer.add(min_observation)
    memory_buffer.add(max_observation)
    memory_buffer.add(mid_observation)
    /* Synthetic Observation Setup                          */
    synthetic_min_observation = min_obervation with
      reward multiplied by −1
    synthetic_max_observation = max_obervation with
      reward multiplied by −1
    synthetic_mid_observation = mid_obervation with
      reward multiplied by −1
    /* Add synthetic observations to memory buffer          */
    memory_buffer.add(synthetic_min_observation)
    memory_buffer.add(synthetic_max_observation)
    memory_buffer.add(synthetic_mid_observation)
```

4 EXPERIMENT AND RESULT

4.1 Implementation

The implementation of the game is done using the No Limit Texas Holdem Poker and Deep Q Agent of the RLCard [6]. The agent provided by the RLCard is taken as the base agent and we have implemented our methods on top of this structure.

4.2 Result and Evaluation

To evaluate our methods, we decided to train each method against a random agent. Later we have made all three of our agents to play against the base agent we are trying to improve. Result we have obtained given below:

Method 1: Figure 1 represents the comparable distribution of rewards for the base agent and the agent created using method 1 where both agents are trained for 100000 episodes. It can easily be observable that the reward graph for our method is going higher when for the base agent it is going lower at the end of the training session.

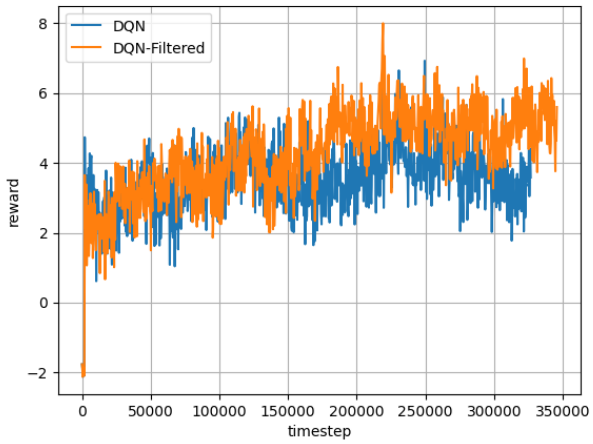


Figure 1: Method 01 - Train Agent

To observe the actual effect of the training we made both agents play with each other. Figure 2 represents the reward distribution for the tournament of 1000 games between the two agents. From figure 2, it is observable that our reward graph is always above the base agent. Within the 1000 games our agent won 781 times which is significantly better for the agents trained the same amount of time.

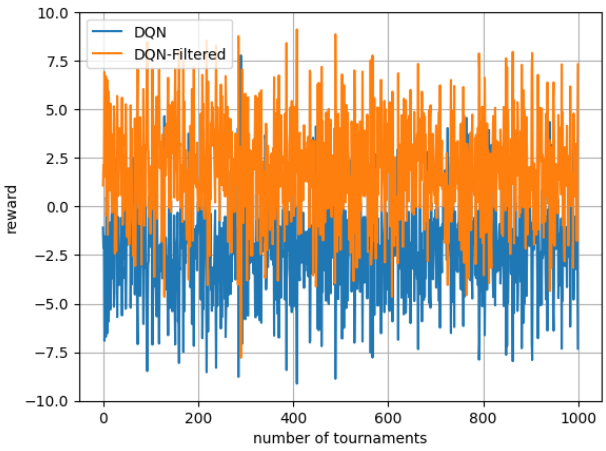


Figure 2: Method 01 - Game Play

Method 2: Figure 3 represents the comparable distribution of rewards for the base agent and the agent created using method 2 where both agents are trained for 100000 episodes. Apparently it is observable that the reward graph for our method is going far below then the base agent here during the training session.

However when we arranged 1000 tournaments between the two trained agents, out method won 661 times out of 1000 times, implying our model is better trained. The reward distribution is shown in Figure 4.

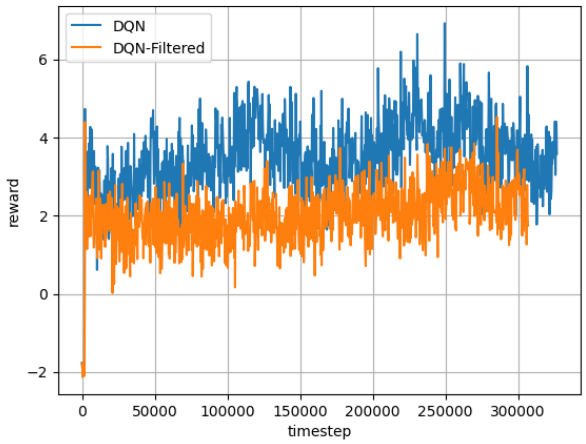


Figure 3: Method 02 - Train Agent

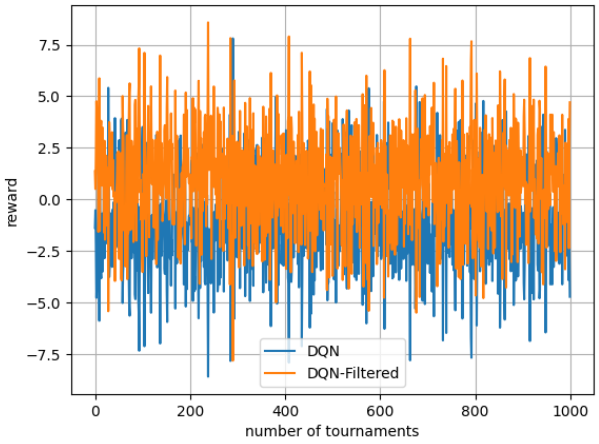


Figure 4: Method 02 - Game Play

Method 3: Figure 5 represents the comparable distribution of rewards for the base agent and the agent created using method 3 where both agents are trained for 100000 episodes. Apparently it is observable that the reward graph for our method is going far below then the base agent here during the training session.

However when we arranged 1000 tournaments between the two trained agents, our method won 673 times out of 1000 times, implying our model is better trained. The reward distribution is shown in Figure 6.

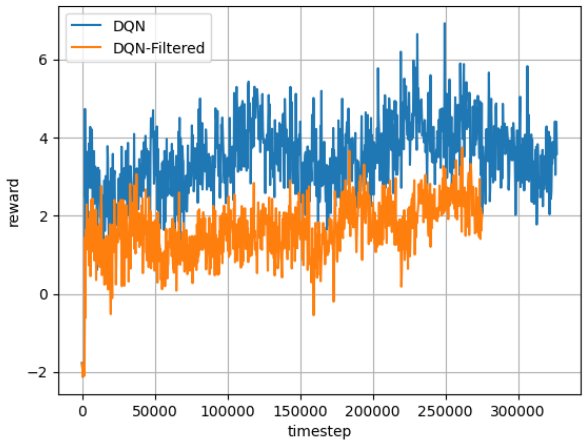


Figure 5: Method 03 - Train Agent

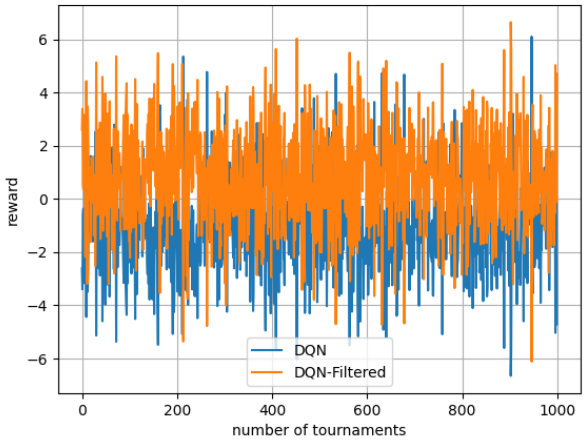


Figure 6: Method 03 - Game Play

5 RELATED WORK

A number of works already have been done to improve the training procedure of deep reinforcement learning using experience replay. Pilachu et. al. have introduced synthetic samples in non-deterministic discrete environment to assist the learning procedure [5]. Their idea is adding better synthetic sample in place of misleading sample to get better performance. They have performed evaluation on frozen lake environment.

Schaul et. al. have developed a method to prioritize samples important for training procedure within the replay buffer [3]. They also employed measures to get rid of any probable overfitting or bias due to the new procedure. They evaluated their game in the environment of *Atari* game.

Andrychowicz et. al. have create a novel technique called Hindsight Experience Replay to learn reward that are sparse and binary [1]. They have evaluated their method for a robotic arm.

Zha et. al. have proposed a novel experience replay optimization framework to optimize the cumulative rewards [7]. They have applied their framework on several *OpenAI Gym* environments.

Foerster et. al. tried to stabilise experience replay for deep multi-Agent reinforcement learning by introducing two methods, using a multi-agent variant of importance sampling to naturally decay obsolete data, and conditioning each agent's value function on a finger print that disambiguates the age of the data sampled from the replay memory [2]. They evaluated their method on a decentralised variant of StarCraft unit micro-management environment.

6 CONCLUSION

In this work, we consider tuning the experience replay buffer to evaluate how optimize a trained agent can behave after introducing new filtering techniques and modification in the memory buffer. From the evaluation of game play between the base agent from *rlcard* and the agent trained based on our modified replay buffer, we find our agents playing the game better each time with greater rewards. Therefore, we can infer that, all three of our proposed methods train an agent better in terms of a partially observable game like no-limit Texas Hold'em Poker game. However, our work does not comprise the evaluation of our methodologies on any other partially observable game. Therefore, in future, we plan to consider tuning and evaluating experience replay as a significant research direction and extend our work thereby for different partially observable games.

REFERENCES

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in neural information processing systems*, pages 5048–5058, 2017.

[2] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017.

[3] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[4] Eilon Solan and Nicolas Vieille. Stochastic games. *Proceedings of the National Academy of Sciences*, 112(45):13743–13746, 2015.

[5] Wenzel Baron Pilar von Pilchau, Anthony Stein, and Jörg Hähner. Bootstrapping a dqn replay memory with synthetic experiences. *arXiv preprint arXiv:2002.01370*, 2020.

[6] Daochen Zha, Kwei-Herng Lai, Yuanpu Cao, Songyi Huang, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A toolkit for reinforcement learning in card games. *arXiv preprint arXiv:1910.04376*, 2019.

[7] Daochen Zha, Kwei-Herng Lai, Kaixiong Zhou, and Xia Hu. Experience replay optimization. *arXiv preprint arXiv:1906.08387*, 2019.

[8] Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008.

A APPENDIX

The used codes of our work is attached in a zip file. Details of the work is described in the "`readme.txt`" file.