Module: Designing Intelligent Agents (COMP3071)
Name: Soo Ying Rui
Student ID: 20297542
Deadline: 17 May 202

Research Title:

Training a Poker Intelligence Agent Using Deep Q-Network Against in a No-Limit Texas Hold'em Environment

Table of Contents

1. Introduction

Poker agent have been performing well against professional agent such as Libratus and DeepStack. However, those two agents can only be applied to two player pokers game. They can’t adapt to multiplayer setting poker game. Therefore, the objective of this research is to create a poker agent and train it to adapt to multiplayer setting poker game No-Limit Texas Hold’em game.

2. Literature Review

2.1 DeepStack

DeepStack is a poker agent that is designed to play poker, which is a complex game of imperfect information that requires players to make decisions based on incomplete and uncertain information about their opponent’s cards and intentions. Specifically, it is designed to play Heads-Up No Limit Texas Hold’em which is a one-on-one poker game without any betting limit. DeepStack uses a combination of supervised learning method called Neural Network and reinforcement learning method called Counterfactual Regret Minimization (CFR) to train and play poker. DeepStack algorithm composed of three main components which are sound local strategy computation for the current public state, depth-limited lookahead using a learned value function to avoid reasoning to the end of the game, and a restricted set of lookahead actions (Moravčík et al., 2017).

Sound Local Strategy Computation: This component is responsible for computing the optimal strategy for the current public state of the game. This involves training a feed forward neural network on a large dataset of previous poker games to allow DeepStack to approximate the value function and predict the value of each possible action. After that, it uses CFR to iteratively improve the strategy based on the outcomes of self-play games. This component only needs minimal memory of how and why it acted to reach the current public state, making it an efficient and effective technique for playing poker.

Depth-limited Lookahead: This component involves using a learned value function to evaluate the strength of each possible action without reasoning all the way to the end of the game tree. This is a more efficient approach than considering all possible outcomes of the game, and allows DeepStack to make quicker decisions in real-time. By using a learned value function, DeepStack is able to approximate the strength of a particular action and use this information to make informed decisions in the game. The lookahead is depth-limited, meaning it only considers a certain number of future actions, which helps to further improve its efficiency.

Restricted Set of Lookahead Actions: This component involves limiting the number of possible actions that can be considered during the lookahead phase. This is done to reduce the size of the game tree and make the decision-making process more efficient. Instead of considering all possible actions, DeepStack selects a subset of the most promising actions based on their estimated value. This component allows DeepStack to focus on the most relevant actions and make more accurate predictions, without wasting computational resources on actions that are unlikely to be optimal. By using a restricted set of lookahead actions, DeepStack is able to strike a balance between accuracy and efficiency in its decision-making process.

Weakness: The main weakness of DeepStack is that it only applies to Heads-Up No limit Texas Hold’em game, which means it only work on two player poker game. This poker agent is used to train in two player poker game. If it is thrown into a multiplayer poker game, the poker agent will need to be retrained and adapt to the multiplayer settings. Another weakness is that even though DeepStack performs well against human opponents in a controlled environment, its performance in real-world scenarios with incomplete or imperfect information is still an open research question.

2.2 Libratus

Libratus is a poker agent that was designed to play the game Heads-Up No-Limit Texas Hold'em poker. It primarily uses the reinforcement learning method called Counterfactual Regret Minimization (CFR) to train its strategy for playing poker. However, it also incorporates several other techniques, such as game theory, action abstraction, information abstraction, and subgame solving, to improve its performance in the game. Libratus features three main modules, and is powered by new algorithms in each of the three. The three modules are computing approximate Nash Equilibrium strategies before the event, subgame solving during play and improving Libratus's own strategy to play even closer to equilibrium based on what holes the opponents have been able to identify and exploit (Brown & Sandholm, 2017).

Abstraction and Equilibrium Finding: In no-limit Texas Hold'em, bet sizes that differ by just one dollar cause different game states and it is quite costly and wasteful to construct a new betting strategy for a single-dollar difference in the bet. To address these challenges, Libratus uses a technique called abstraction, which groups similar bets and other actions into categories known as blueprint. In a blueprint, similar bets will be treated as the same and so are similar card combinations (Zhu, 2021). The blueprint is significantly smaller than the possible number of states in the game, making it possible for Libratus to solve the game using CFR, an iterative, linear-time algorithm that solves for Nash Equilibria in extensive form games. In CFR, two agents play against each other and try to minimize their own counterfactual regret with respect to the other agent's current strategy. Instead of enumerating every leaf node of the game tree, Libratus uses a Monte Carlo-based variant of CFR to samples the game tree to get an approximate return for the subgame. This approach allows the poker agent to efficiently solve the game using smaller number of computations (Zhu, 2021).

Nested Safe Subgame Solving: This algorithm solves a finer-grained abstraction of the remaining game in the third round using a subgame-solving technique. The subgame-solving creates and solves a new subgame every time an opponent chooses an action that is not in the finer-grained abstraction. This allows Libratus to avoid the rounding error due to action translation and leads to lower exploitability. Another novel aspect of the subgame solver is that it guarantees that the solution is no worse than the precomputed equilibrium approximation, taking into account the opponent's mistakes in the hand so far to enlarge the strategy polytope that can be safely optimized over. Additionally, Libratus changes its action abstraction in each subgame to make it more difficult for opponents to adapt (Brown & Sandholm, 2017).

Self-Improvement: The third module of Libratus focuses on improving its strategy on the first two betting rounds by using a dense action abstraction. However, if an opponent's bet is not in the abstraction, there may be slight errors. To mitigate this, Libratus selects a few actions to add to the abstraction based on the opponent's frequently chosen actions and their distance from the nearest action in the abstraction. A strategy is then computed for the selected action, and if the opponent selects that action or a nearby one, Libratus will use the newly solved subgame strategy (Brown & Sandholm, 2017).

Weakness: Libratus also have the same weakness as DeepStack which is it only apply to Heads-Up No limit Texas Hold'em game. It is not adapted to multiplayer game. Therefore, if the poker agent is thrown into a multiplayer poker game, the poker agent will lose and need to be retrained and adapt to the multiplayer settings.

3. Experiment

3.1 Poker Environment

The environment that is used to train the agent is a poker environment. The environment is used to simulate the game of No-Limit Texas Hold'em. In No-Limit Texas Hold'em, each player is dealt two private cards and then five community cards will gradually be revealed in rounds of betting. Three community cards will be revealed during Flop and one community card will be revealed during Turn and River. The goal is to make the best possible five-card hand using any combination of the two private cards and the five community cards. The state of the game is represented as a dictionary of information that includes the private cards of each player, the public cards on the table, and the amount of chips that each player has. The actions available to the agent are represented as integers, which correspond to different actions such as folding, calling, or raising. The observation space is a dictionary that contains the same information as the game state, but with some of the opponent's private cards hidden.

3.2 Deep Q-Network Agent

The poker agent that is used to train is the Deep Q-Network (DQN) agent where it uses DQN method to train and play the game. The architecture of the DQN agent consists of four main components which are a deep neural network that approximates the Q-value function, an experience replay buffer that stores past experiences, a target network that stabilises the learning process and an epsilon-greedy strategy to balance exploration and exploitation.

Deep Neural Network: The deep neural network is used to approximate the Q-value function, which estimates the expected cumulative reward for taking a particular action in a given state. The neural network consists of two fully connected layers with ReLU activation functions. The input to the network is a vector of size 118, which represents the state of the game. The output of the network is a vector of size 3, which represents the Q-values for the three possible actions which are fold, call, and raise. During training, the network parameters are updated using the Adam optimizer.

Experience Replay Buffer: This component is a data structure that stores the agent's experiences in the form of (state, action, reward, next state) tuples. During training, the agent randomly samples a mini-batch of experiences from the buffer and uses them to update the neural network parameters. This helps to break the correlation between consecutive experiences and improve the stability of the learning process.

Target Network: The target network is a copy of the Q-network that is used to generate the target Q-values during training. The parameters of the target network are updated periodically, typically every few thousand steps, to keep it in sync with the Q-network. The purpose of using a target network is to reduce the variance of the Q-value estimates and make the learning process more stable.

Epsilon-Greedy Exploration: The agent uses an epsilon-greedy exploration strategy to balance exploration and exploitation during training. At each time step, the agent selects a random action with probability epsilon and selects the action with the highest Q-value with probability 1-epsilon.

3.3 Training and Evaluation

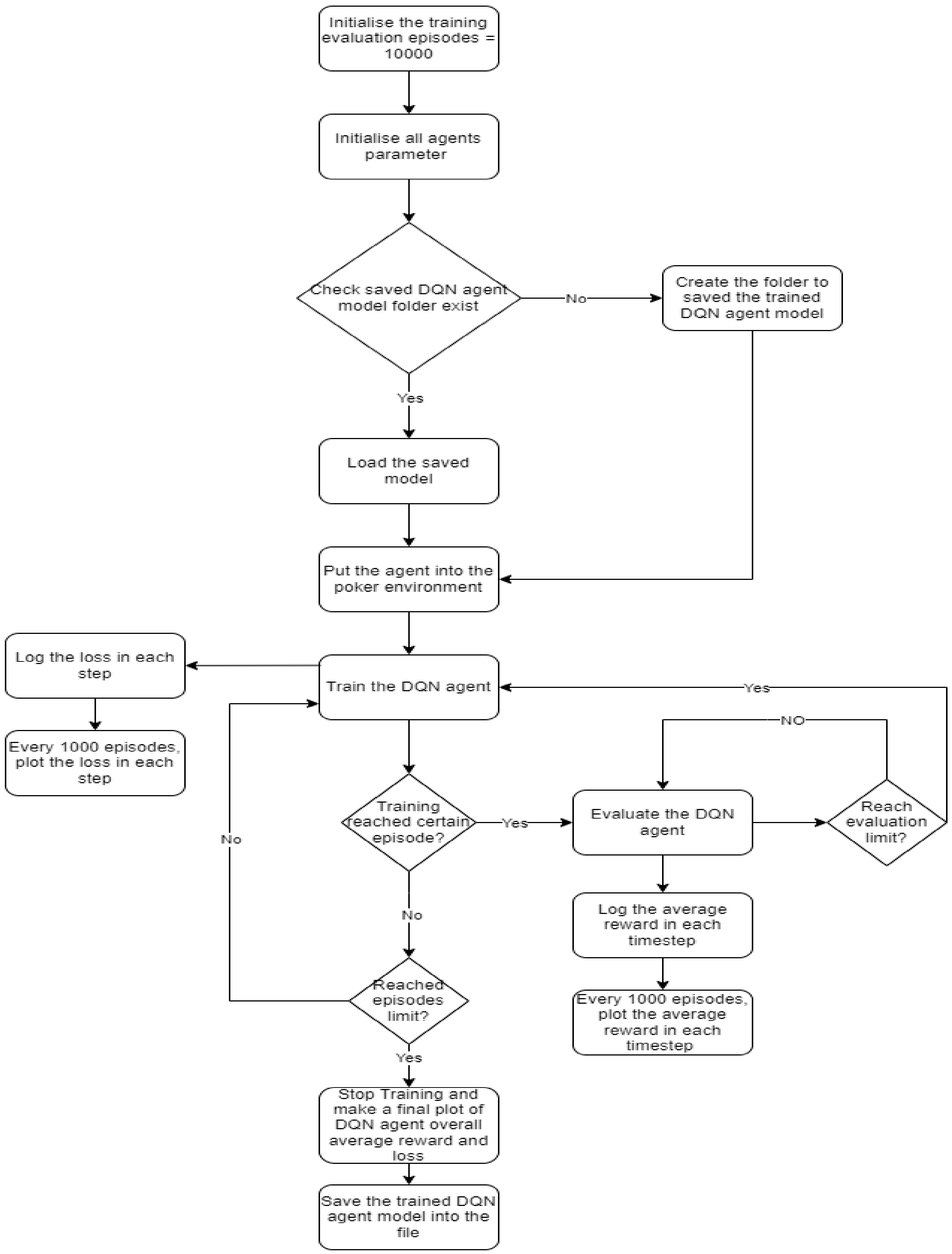
3.3.1 Training and Evaluation Opponents

The DQN agent is supposed to be train and adapt to multiplayer poker game. Therefore, the DQN agent is training and evaluating against two other agents which are Random agent and Neural Fictitious Self-Play (NFSP) agent.

Random Agent: This agent is a simple baseline agent that acts randomly in the poker environment. At each decision point in the game, the Random Agent chooses a legal action uniformly at random from the set of available actions. This agent is a very basic implementation of an agent and does not take into account the underlying game structure or the strategies of the other players in the game. The reason of using this agent to train and evaluate the DQN agent is to simulate random poker players in real life. Especially those players who play without patterns such as making random bet.

NFSP Agent: This agent combines both neural networks and fictitious self-play to learn a Nash equilibrium strategy in poker games. It has two neural networks, one for the opponent model and one for the strategy, that are updated simultaneously. During self-play, the agent trains its strategy network using the opponent model network to generate fictitious counterfactual regrets. These regrets are then used to update the strategy network using a variant of the Q-learning algorithm. The opponent model network is updated by playing against the current strategy network and using the outcome to update the weights of the opponent model network using a variant of the Q-learning algorithm. NFSP agent also uses a modified version of DQN algorithm to train its strategy network, which includes a target network to stabilise training and an epsilon-greedy exploration policy. The reason of using this agent to train and evaluate the DQN agent is to simulate professional poker players in real life.

3.3.2 Training and Evaluation Procedure



Initialise the training evaluation episodes = 10000
Initialise all agents parameter
Check saved DQN agent model folder exist
No
Create the folder to saved the trained DQN agent model
Yes
Load the saved model
Put the agent into the poker environment
Log the loss in each step
Train the DQN agent
Yes
Every 1000 episodes, plot the loss in each step
NO
No
Training reached certain episode?
Yes
Evaluate the DQN agent
Reach evaluation limit?
No
Log the average reward in each timestep
Reached episodes limit?
Every 1000 episodes, plot the average reward in each timestep
Yes
Stop Training and make a final plot of DQN agent overall average reward and loss
Save the trained DQN agent model into the file

Figure 1: DQN Agent Training and Evaluation Procedure

Figure 1 is the entire procedure of training and evaluating the DQN agent. The agent will be training for 10000 episodes, which means the agent will play 10000 times of the poker game. During the training, the agent loss in each step will be recorded into excel and text file. The files will be saved into the "Experiment" folder. Every 100 episodes, the agent evaluation will be started. In each evaluation, the DQN agent will have to play 1000 games of poker. During evaluation, the agent average reward in each timestep will be recorded into excel and text file. The files will then be stored into the "Experiment" folder. For every 1000 episodes, the agent loss in each step and average reward in each timestep will be plotted into line graph and saved into the "Experiment" folder. After the agent training is done, the DQN agent model will be saved and it will be loaded back when we want to train the DQN agent again. After the training, the overall agent loss and average reward will be plotted into line graph.

3.4 Hyperparameters and Training and Evaluation Amount

Hyperparameter	Value
action_num	env.action_num
replay_memory_size	1000000
replay_memory_init_size	1000
update_target_estimator_every	1000
epsilon_start	1.0
epsilon_end	0.1
epsilon_decay_steps	20000
norm_step	100
state_shape	env.state_shape
mlp_layers	[512, 512]
batch_size	512
learning_rate	0.0005
discount_factor	0.95

Table 1: DQN Agent Hyperparameter

Table 1 is the value of DQN agent hyperparameter. ***"action_num"*** is the number of possible actions that the agent can take in the environment. ***"replay_memory_size"*** is the maximum size of the replay buffer. ***"replay_memory_init_size"*** is the number of initial experiences to add to the replay buffer before training begins. ***"update_target_estimator_every"*** is the frequency at which the target network is updated with the weights of the primary network. ***"epsilon_start"*** is the initial value of epsilon in the epsilon-greedy exploration strategy. ***"epsilon_end"*** is the final value of epsilon in the epsilon-greedy exploration strategy. ***"epsilon_decay_steps"*** is the number of steps over which epsilon is linearly decayed from "epsilon_start" to "epsilon-end". ***"norm_step"*** is the frequency at which the weights of the network are normalised to help stabilise the learning process. ***"state_shape"*** is the shape of the state representation used by the agent. ***"mlp_layers"*** is the architecture of the neural network used to approximate the Q-values. As you can see the table above, the neural network has two layers with 512 nodes in each layer. ***"batch_size"*** is the size of mini-batches used for training the network. ***"learning_rate"*** is used by the optimiser to update the network weights during training. ***"discount_factor"*** is the discount factor for future rewards. This determines the weight given to future rewards in the agent's learning process.

Iteration	Amount
evaluate_every	100
save_plot_every	1000
evaluate_num	1000
episode_num	10000

Table 2: DQN Training and Evaluation Amount

Table 2 is the training and evaluation amount of the DQN agent. ***"episode_num"*** is the total number of episodes used for training the agent. ***"evaluate_num"*** is the number of games played during each evaluation of the agent. ***"save_plot_every"*** is the frequency (in episodes) at which a plot of the agent's performance is saved. ***"evaluate_every"*** is the frequency (in episodes) at which the agent is evaluated.

3.5 Metrices

To measure the performance of the DQN agent, "step", "loss", "timestep" and "average reward" are used.

Step: This measure keeps track of the number of steps that the agent has taken in the environment. It is incremented by 1 for each transition fed into the agent memory during training.

Loss: The loss is a measure of how much the Q-value estimates produced by the agent differ from the expected Q-value. The goal is to minimise the loss over time, which implies better Q-value estimates and better performance in the poker environment.

Timestep: Timestep refers to a discrete unit of time that defines a moment in the interaction between an agent and an environment. At each timestep, the agent observes the current state of the environment, selects an action, and sends it to the environment. The environment then transitions to a new state and returns a reward to the agent.

Average Reward: This measure is the average reward of the total game in each evaluation.

The "loss" is plotted against "step" and the "average reward" is plotted against the "timestep".

4. Result

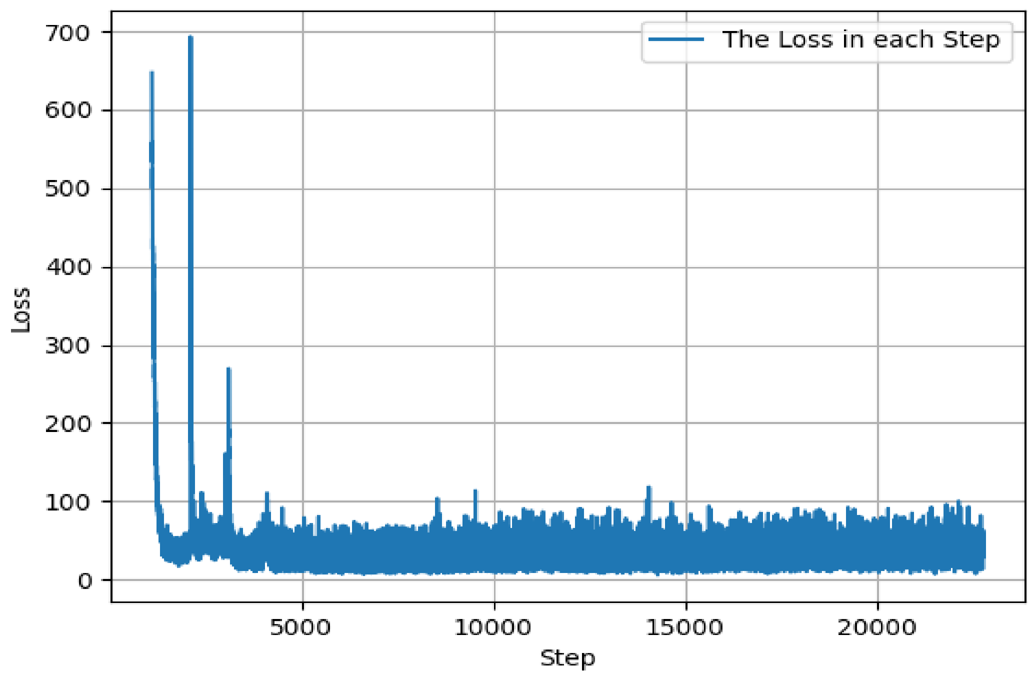


Figure 2: Overall DQN Agent Loss in Each Step

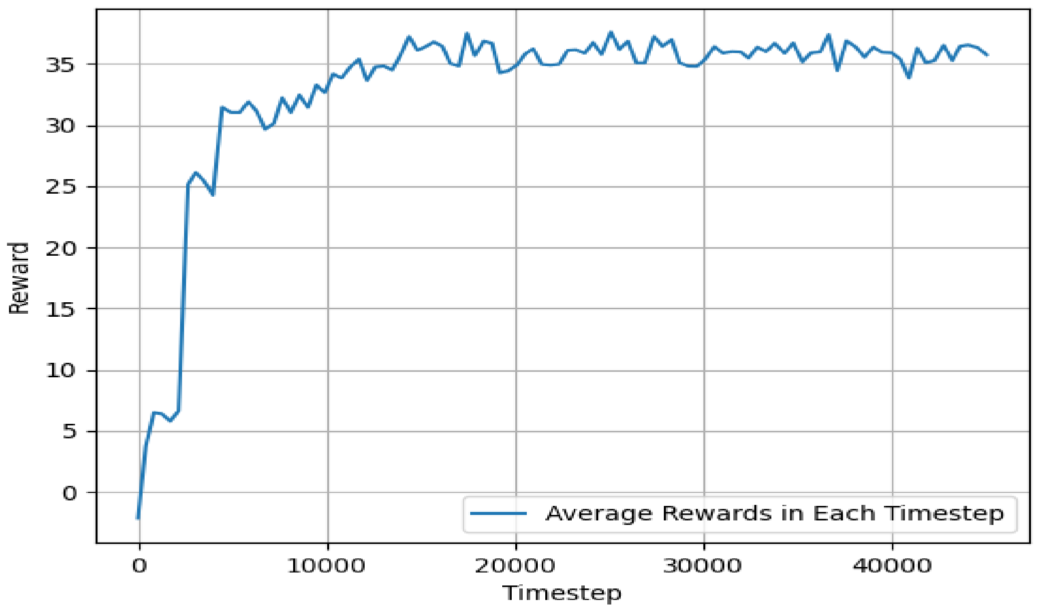


Figure 3: Overall DQN Agent Average Reward in Each Timestep

5. Discussion

5.1 Result

The loss of DQN agent measurement is done during training and the average reward measurement is done during evaluation. The poker agent took a total of 22763 steps and 45051 timestep to finish training and evaluation for 10000 episodes. After the training and evaluation, we can see the significant increase of DQN agent performance.

In **figure 2**, there are two notable spikes in the beginning of training. The first spike in the loss is due to the reason of DQN agent new to the environment. In the early stages of training, the agent is new and have little experience in playing the poker game. Therefore, the agents will starts initialising its weights randomly, which lead to a wide range of Q-value estimates for each state-action pair. The Q-value estimates can be highly uncertain and volatile which causes a sudden spike in the loss, as the agent struggles to learn an accurate representation of the environment. Furthermore, the exploration and exploitation trade off can be challenging for the agent in the beginning of training. The agent is trying to balance between exploring new actions to gather more information about the environment and exploiting the current best action to minimise the loss. This can lead to more unstable learning process in the beginning, which also causes a sudden spike in the loss.

The second spike in loss is caused by the non-stationarity of the environment. In a poker environment, the actions of the opponent players are not fixed and may change over time. This led to a non-stationary environment, where the optimal actions of the agent may change over time, which causes the loss to spike as the agent adjusts its policy to the changing environment.

After the two significant spikes in the loss, the loss of the agent gradually going down and eventually stabilise and remain low. This indicates that the poker agent is able to accurately predict the expected Q-values and has learned to perform optimally in the poker environment.

In **figure 3**, the average reward of the agent in the early stages of evaluation is low. The reason for this is the DQN agent is new to the game of poker. It may also encounter situations that it has not experience during training. This led to low average reward in the early stages of evaluation, as the agent needs to learn from these new experiences to improve its performance.

Despite the fact that the average reward of the DQN agent start from low, the average reward gradually increase overtime and eventually maintain high level. This indicates that the agent is learning and has learned to take actions that lead to better outcomes in the poker game.

For every 1000 episodes, I have plot and saved the loss and average reward of the DQN agent. The graph will be shown in **Appendix A** and **Appendix B**. The detail values of loss and average reward in each step and timestep is put in an excel file and text file and both files are stored in the "experiment" folder in the code file.

5.2 Benefits and Limitation

The main benefits of my approach in training and evaluating the agent is that it can adapt in a multiplayer poker game. I train and evaluate the DQN agent against two agents in order for the DQN agent to adapt in a multiplayer setting. I chose Random agent and NFSP agent as training and evaluating opponent is to simulate real life situation. In real life, there are all kinds of poker players, such as random player who play poker with making random choices and professional player who play poker smartly. I want the random agent to represent the player who play the poker game randomly and NFSP agent represent the professional player who play poker smartly. This training is to allow the agent to adapt to all kinds of player in a multiplayer poker game.

Even though there are benefits training the agent against other agents, there's still have limitation to this approach. I have only trained the agent against two types of players. In the real world, each player has their own playing style. Due to the limitation of my computer power, I'm not able to include all types of players into my agent training. Therefore, the agent might not perform well when it meets some player who have their own style of playing poker.

6. Conclusion

In conclusion, the DQN agent has shown great promise in its ability to learn and adapt to different playing styles and strategies. Through the use of reinforcement learning and neural network, the agent is able to learn from its own experiences and improve its performance over time. There are still limitations to the technology However, in future, I will do more extensive training and add more multiplayer setting such as adding all types of players into the training.

7. Reference

1. Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., & Bowling, M. (2017). DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. Science, 356(6337), 508–513. https://doi.org/10.1126/science.aam6960
2. Brown, N., & Sandholm, T. (2017). Libratus: The superhuman AI for No-Limit Poker. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*. https://doi.org/10.24963/ijcai.2017/772
3. Zhu, J. (2021, December 30). *Libratus: The world's best poker player*. The Gradient. https://thegradient.pub/libratus-poker/

Appendix A: Loss Result

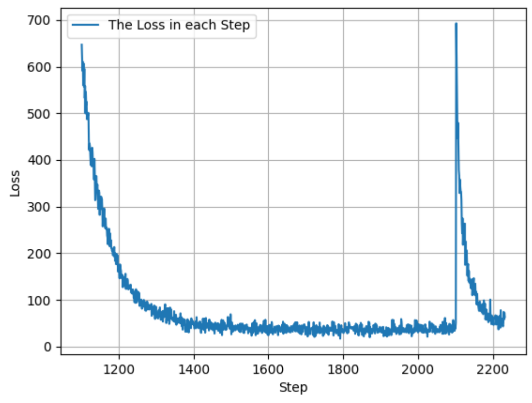


Figure 4: Episode 1 – 1000 Agent Loss in Each Step

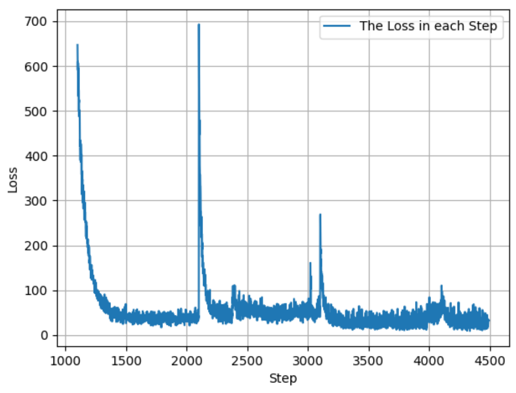


Figure 5: Episode 1001 – 2000 Agent Loss in Each Step

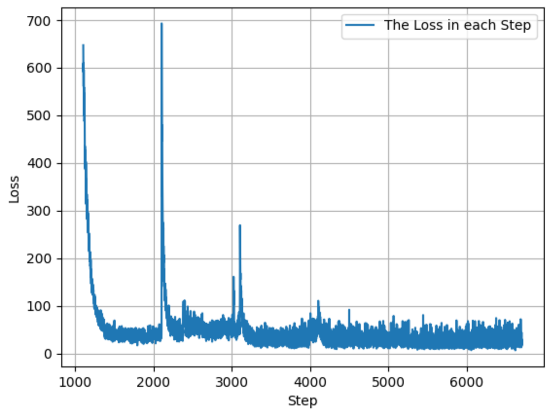


Figure 6: Episode 2001 – 3000 Agent Loss in Each Step

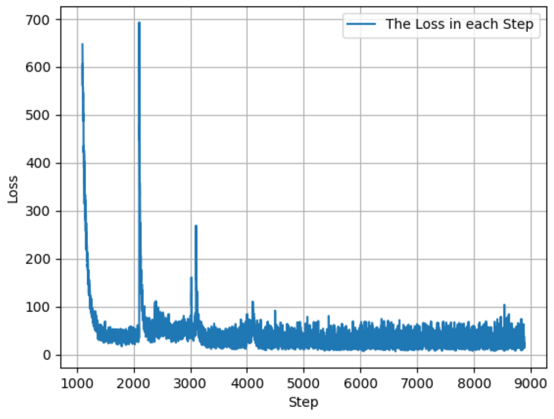


Figure 7: Episode 3001 – 4000 Agent Loss in Each Step

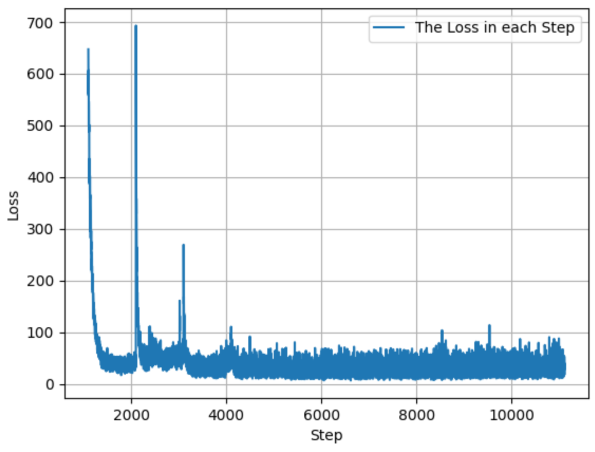


Figure 8: Episode 4001 – 5000 Agent Loss in Each Step

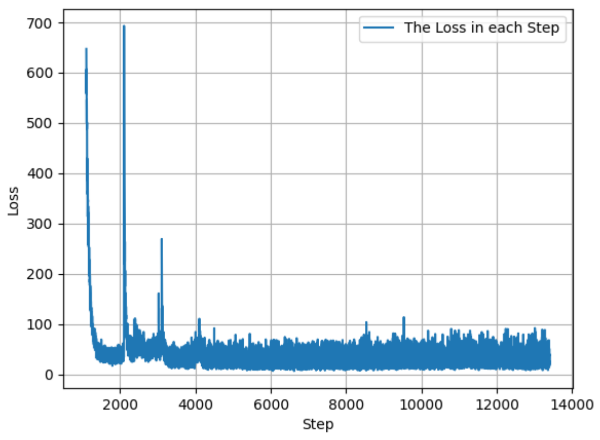


Figure 9: Episode 5001 – 6000 Agent Loss in Each Step

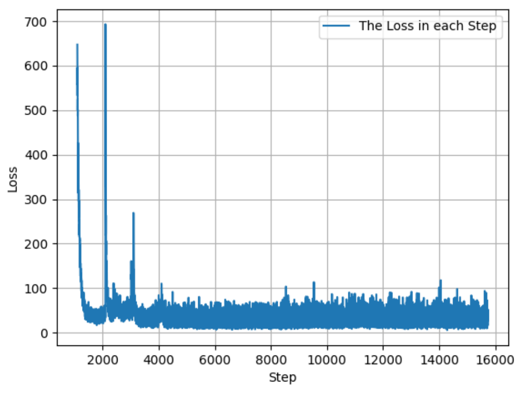


Figure 10: Episode 6001 – 7000 Agent Loss in Each Step

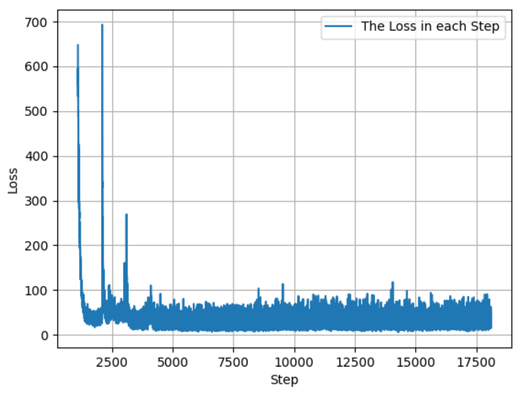


Figure 11: Episode 7001 – 8000 Agent Loss in Each Step

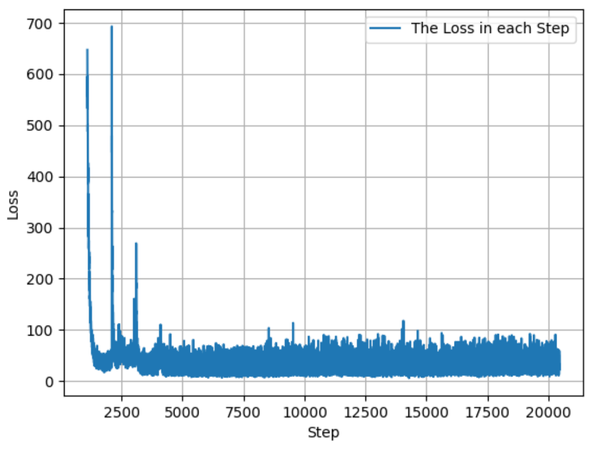


Figure 12: Episode 8001 – 9000 Agent Loss in Each Step

Appendix B: Average Reward Result

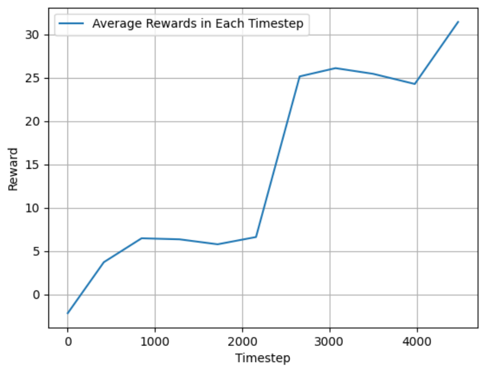


Figure 13: Episode 1 – 1000 Agent Average Reward in Each Timestep

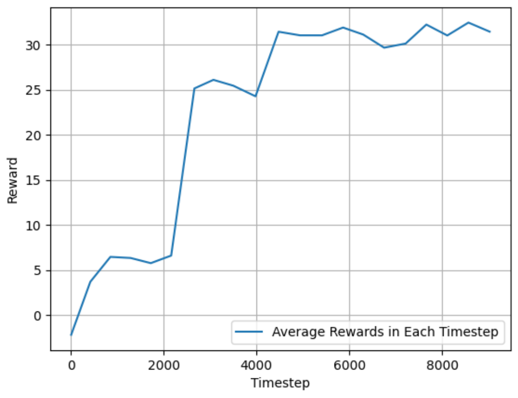


Figure 14: Episode 1001 – 2000 Agent Average Reward in Each Timestep

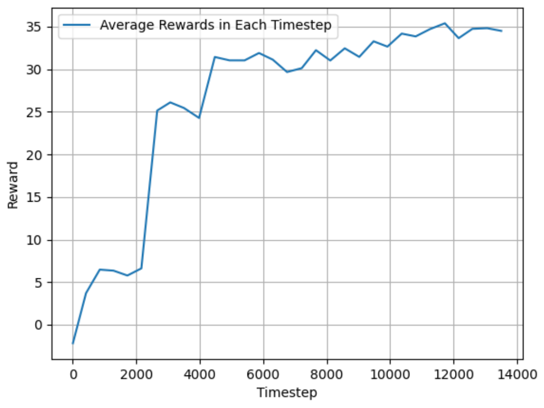


Figure 15: Episode 2001 – 3000 Agent Average Reward in Each Timestep

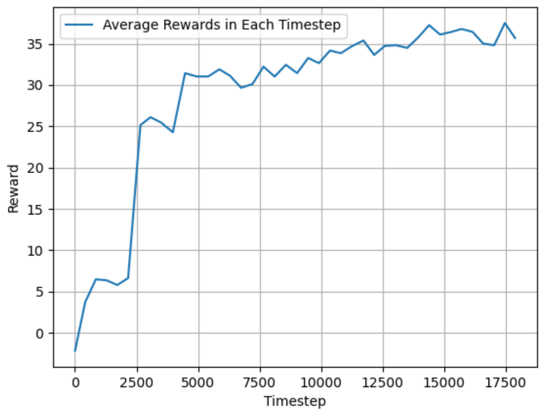


Figure 16: Episode 3001 – 4000 Agent Average Reward in Each Timestep

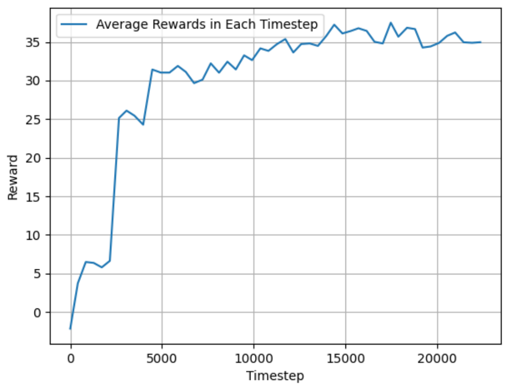


Figure 17: Episode 4001 – 5000 Agent Average Reward in Each Timestep

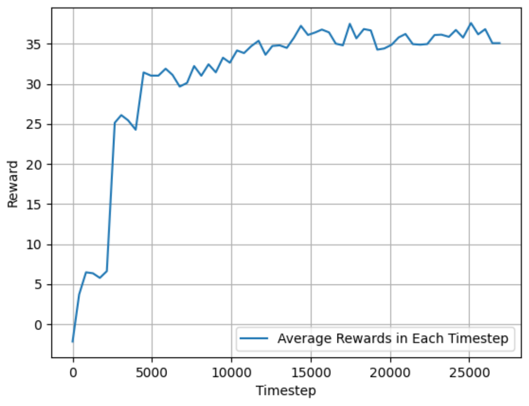


Figure 18: Episode 5001 – 6000 Agent Average Reward in Each Timestep

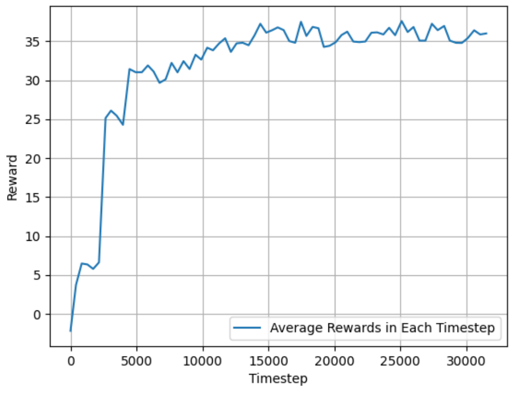


Figure 19: Episode 6001 – 7000 Agent Average Reward in Each Timestep

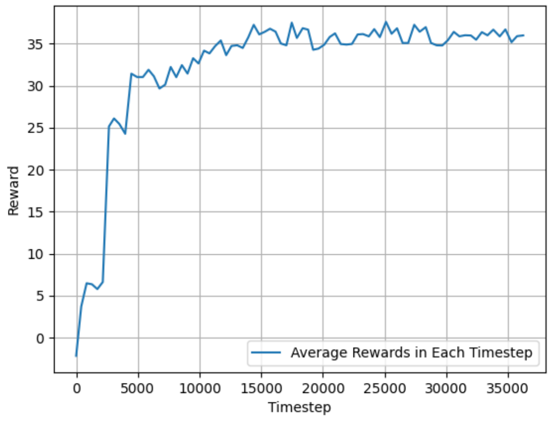


Figure 20: Episode 7001 – 8000 Agent Average Reward in Each Timestep

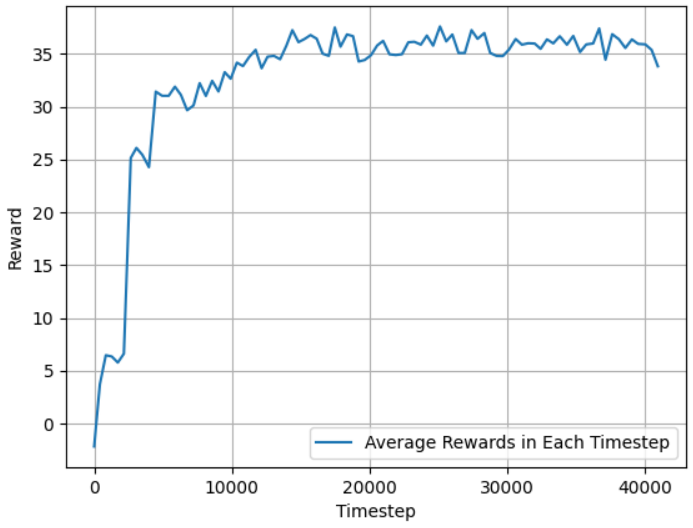


Figure 21: Episode 8001 – 9000 Agent Average Reward in Each Timestep