Deep Reinforcement Learning for Poker: A DQN-Based Approach

Tobin Lee
Electrical and Computer Engineering C147
University of California, Los Angeles
Los Angeles California, 90024
`tobinl09@g.ucla.edu`

Abstract

In this project, we implemented a deep reinforcement learning agent using Deep Q-Networks (DQN) to play heads-up No-Limit Texas Hold'em poker. Our approach involved modeling the poker environment using PyPokerEngine, training two agents against each other, and optimizing decision-making through replay memory and target networks. We experimented with various hyperparameters and analyzed the impact of opponent modeling by encoding previous opponent actions into the state representation. Our preliminary results indicate that the agent is capable of learning basic poker strategies. However, further tuning is needed for strong generalization. We discuss key insights, limitations, and future improvements.

1 Introduction

Poker is a challenging game for artificial intelligence due to its imperfect information and high variance. Traditional rule-based systems struggle to generalize effectively, while recent advances in reinforcement learning (e.g., DeepStack) have shown promising results in handling strategic decision-making in poker. In this work, I will employ a deep Q-learning approach to train an AI agent for heads-up No-Limit Texas Hold'em using a customized environment built with PyPokerEngine. The primary goal is to evaluate whether a DQN-based approach can learn competitive poker strategies through self-play and experience replay.

2 Methods

2.1 Environment and state representation

I initially planned to use Q-learning to map each state to an action (e.g., Fold, Call, Raise). Simply considering the 1326 unique combinations of starting hands and five streets (preflop, flop, turn, river, showdown) which can have various amount of actions I soon realized that hard coding every possible state that may occur in poker would be near impossible. Instead I decided to implement a deep Q-learning approach to help calculate approximate Q-values based on similar states. With this approach my agent would no longer need to map every unique state to a Q-value.

I implemented the poker game environment using PyPokerEngine, which simulates poker rounds and maintains game state. The state representation includes:

- Hand strength estimation using Monte Carlo simulations.
- Pot size normalized by the initial stack.
- Agent stack size normalized by the initial stack.
- Big blind position as a binary feature.

- Opponent's last action (encoded as an integer).
- Opponent's last bet size normalized by the stack.

2.2 Deep q-network architecture

The Deep Q-Networks model consists of three fully connected layers:

- Input: 6-dimensional state vector.
- Hidden layers: Two layers of 128 ReLU-activated neurons.
- Output: Q-values for three actions: Fold, Call, Raise.

2.3 Experience replay

Experience replay is a crucial technique in Deep Q-Networks that improves learning efficiency, stability, and generalization. In poker, where decision-making involves long-term strategy and imperfect information, replay memory allows the agent to store past experiences. Each consists of a state, action, reward, next state, and terminal flag. Instead of updating the agent using only the most recent experience, the agent randomly samples past experiences for training, reducing correlations between consecutive states and improving sample efficiency. This approach stabilizes learning by preventing drastic updates to Q-values, which is especially important in poker, where rewards are often not immediate. Along with the replay memory the model uses the Bellman equation to periodically update the target network from the policy network's weights.

2.4 Reward function

The reward function is designed to guide the poker agent's learning process by incorporating multiple factors that influence strategic decision-making. The primary component of the reward is stack-based reinforcement, where the change in a player's chip count relative to the big blind is used as a fundamental measure of success. Additionally, the function encourages strategic play by providing a small positive reward for checking when it results in seeing the next community card for free, promoting cost-efficient play. To further refine decision-making, hand strength progression is rewarded, incentivizing the agent to take actions that improve its relative position in the game. A win/loss reward is incorporated at the end of each round, granting a significant positive reward for winning and a penalty for losing, reinforcing the importance of long-term strategic planning. Lastly, a folding penalty discourages excessive folding, ensuring that the agent remains competitive rather than passively conceding hands. By combining these elements, the reward function balances aggression, adaptability, and profitability, enabling the agent to develop a well-rounded poker strategy through reinforcement learning. Lastly these reward factors have been normalized to ensure stability in training.

2.5 Training process

We trained two DQN agents against each other over at most 200 episodes. Each episode consisted of at most 100 unique hands. The training process involved two distinct poker agents with of the same model playing each other. The key baseline hyperparameters used were:

- Learning rate: 1×1^{-5}
- Discount factor (γ): 0.99
- Epsilon decay: Over 2000 steps, from 1.0 to 0.1
- Batch size: 128
- Replay memory size: 10,000
- Soft update coefficient (τ): 0.005
- Loss Function: Huber Loss Function

3 Results

3.1 Q-values

Early training results show promising trends in Q-values. While the model lacked extensive training time, Q-values often exhibited convergence to a consistent range or followed a predictable trend. This suggests that, given additional training, Q-values would likely stabilize further, leading to more reliable decision-making. Figure 1 shows that the agent with the most training data has the most stable Q-values as expect.

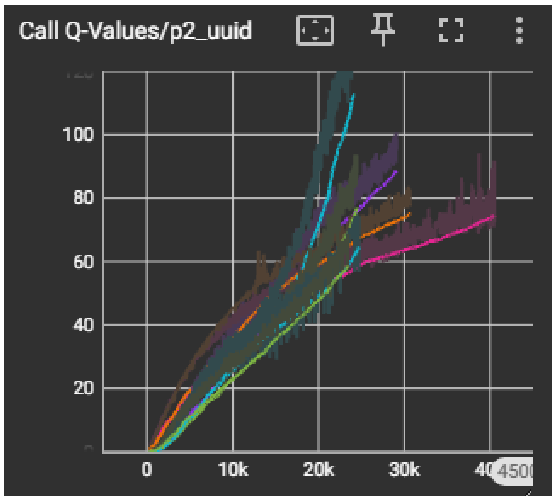

Figure 1: Call Q-Values

3.2 Reward function

Similarly to the Q-Values, our reward function is seen converging to similar values across different training iterations. However, this data does not necessarily mean that the reward function accurately depicts state-to-action values. A scenario such as the reward function biases to only safe actions therefore the reward remains similar across all iterations.

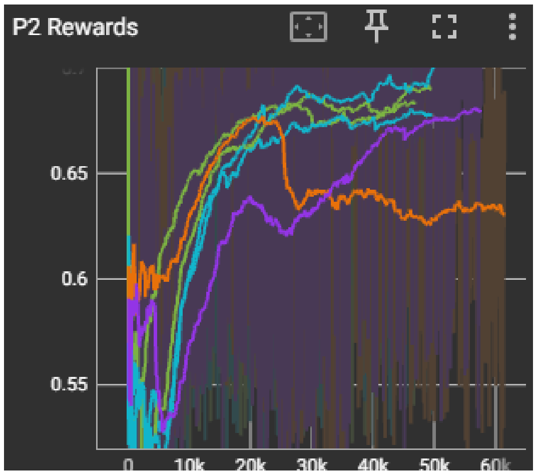

Figure 2: Reward Function

3.3 One agent dominance

Despite both agents being initialized with identical hyperparameters for every training instance, one agent consistently emerged as the dominant player. Even when stack sizes were reset, the superior agent maintained its advantage. This suggests that one agent was able to adapt faster and exploit weaknesses in the opponent.

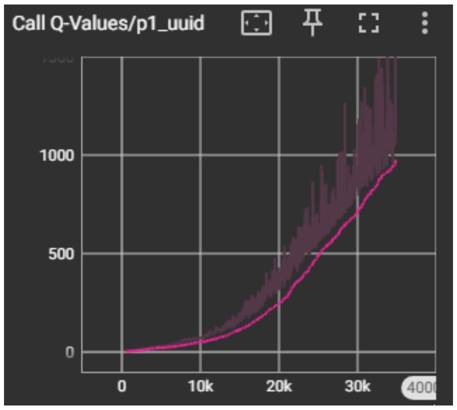

Figure 3: Agent 1 Call Q-Value

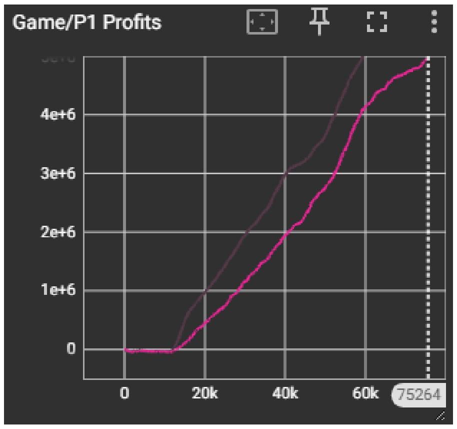

Figure 4: Agent 1 Stack Growth

4 Discussion

4.1 Q-values and reward functions

An issue observed during training was the divergence of Q-values between the two models. One agent's Q-values would occasionally explode, while the other's would converge within a reasonable range. This trend was noticed when one model took a early lead against the other. A agent winning a couple of hands may over-reward a raise while on the other hand, the losing agent keeps folding to preserve their stack, highlighting a key issue in the reward function. If rewards are not proportionally distributed based on action value and long-term gains, the model may assign excessive value to suboptimal moves, causing instability. Due to the difficulty of crafting an optimal reward function in poker, this remains a critical area for improvement. Additionally, limited training data may have contributed to the instability; with extended training iterations, Q-values may eventually settle into a more balanced equilibrium.

4.2 Insights and Observations

Initially, I started out with hard updating my target network for a given amount of steps. The target network abruptly being changed by the policy network causes a high-loss function. However, after changing to a soft update where the target network gets updated slowly by a factor of tau, I greatly reduced the loss function. Figure 5 depicts hard updates, in pink, and soft updates, in green. This stabilized training by preventing sudden changes in the expected Q-Values and helps generalizing more states. The stability in Q-values is of the utmost importance because the agent needs to learn consistently good moves. Having highly varying Q-Values. This instance can been seen when over-betting a low pocket pair into a premium hand like Ace, King. Although pot odd equities may be near 50/50 it doesn't necessarily mean it is a good move, or high in Q-value.

5 Conclusion

As poker is a game where it is best when the opponent has the least knowledge, there are very little datasets containing poker run outs. Due to this factor, we must rely on self-learning algorithms and run validation tests through brute force. This creates an inefficient method to test the model. Using a Deep Q-Network can lead to significant training instability unless the hyperparameters are

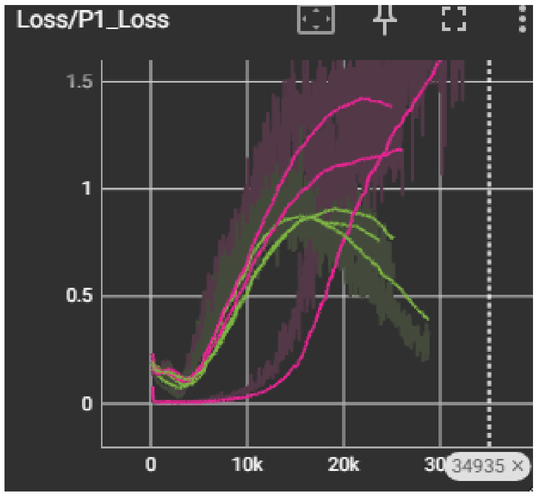

Figure 5: Enter Caption

carefully tuned. However even with a short data set showing the convergences of Q-Values and the loss function demonstrates a stable, rule-based playing model can be created.

References

References follow the acknowledgments in the camera-ready paper. Use unnumbered first-level heading for the references. Any choice of citation style is acceptable as long as you are consistent. It is permissible to reduce the font size to `small` (9 point) when listing the references. Note that the Reference section does not count towards the page limit.

[1] Corino, C. (2024). Building a Deep Q-Network powered Poker Bot - Corino - Medium. [online] Medium. Available at: https://medium.com/@mycorino/building-a-deep-q-network-powered-poker-bot-1a48e296805d [Accessed 15 Mar. 2025].

[2] Feng, Y., Subramanian, S., Wang, H. and Guo, S. (2020). Train a Mario-playing RL Agent — PyTorch Tutorials 1.8.1+cu102 documentation. [online] pytorch.org. Available at: https://pytorch.org/tutorials/intermediate/mario$_rl_tutorial.html.$

[3] Ishikota (2025). Introduction - PyPokerEngine. [online] Github.io. Available at: https://ishikota.github.io/PyPokerEngine/ [Accessed 15 Mar. 2025].

[4] Moravcik, M., Schmid, M., Burch, N., Lisy, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. and Bowling, M. (2017). DeepStack: Expert-Level Artificial Intelligence in Heads-Up No-Limit Poker.

[5] Paszke, A. and Towers, M. (2024). Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.8.0 documentation. [online] pytorch.org. Available at: https://pytorch.org/tutorials/intermediate/reinforcement$_qlearning.html.$

[6]Paszke, A. and Towers, M. (2024). Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.8.0 documentation. [online] pytorch.org. Available at: https://pytorch.org/tutorials/intermediate/reinforcement$_qlearning.html.$