Adaptive Deep Reinforcement Learning: Strategies for Overcoming Individual Poker Styles

Chase Hameetman
Harvard-Westlake School
chasejh1@gmail.com

Abstract - This research paper delves into the development and evaluation of advanced artificial intelligence models in the complex domain of Texas Hold'em poker, focusing on the models' ability to adapt to individual player styles and strategies. The study first involved creating a baseline AI model to mirror an average player's playstyle using Long Short-Term Memory (LSTM) models, followed by training and optimizing a deep reinforcement learning (RL) model through self-play and competitive scenarios against LSTM models. The effectiveness of the RL models, both optimized and unoptimized Deep Q-Networks (DQN), were critically assessed in a simulated poker environment. Testing revealed that the LSTM was able to emulate playstyle with 80% accuracy. Moreover, while the unoptimized DQN models struggled against LSTM opponents, the optimized version exhibited a substantial improvement, highlighting AI's ability to quantify and counter human strategy.

1. Introduction

Poker, a game renowned for its intricate blend of strategy, luck, and skill, necessitates a deep understanding of its complexities to make effective, semi-optimal decisions. Unlike many other games, poker's unpredictability, driven by the inherent factor of luck, adds a layer of complexity that challenges even the most skilled players. However, the realm of artificial intelligence (AI) has seen groundbreaking advancements, particularly in mastering such unpredictable environments. AI has managed to dominate even top-tier poker players, a testament to its evolving capabilities. This dominance is primarily due to a sophisticated strategy known as regret minimization. Through this strategy, AI algorithms make choices based on minimizing future regret, a concept crucial in uncertain environments like poker. For instance, a move causing regret could be folding a winning hand or raising with a losing one. In executing this strategy, these AI bots have showcased exceptional skill in bluffing and overall game sense. Their performance remains consistently high, adhering to an optimal game plan that does not vary according to the skill set or style of the opposing players.

Yet, each poker player brings a unique flavor to the game, characterized by individual playstyles. Some players might have a penchant for folding under pressure, others for bluffing with an air of confidence, or still others for aggressively raising the stakes when they hold a strong hand—or alternatively, maintaining a more conservative approach. These tendencies become particularly evident when one is pitted against amateur or casual players. Unlike seasoned professionals, casual players often do not project their strategies several moves ahead, making their gameplay somewhat unpredictable. This leads to an intriguing consideration in the field of AI. **Can artificial intelligence not only master the probabilistically optimal moves, but also dynamically adapt to individual players' styles, effectively countering their unique gameplay quirks and strategies?** Such an ability would not only demonstrate AI's proficiency in decision-making under uncertainty but also its potential for personalized interaction and adaptability in complex, human-centric environments. This evolution in AI's approach could redefine the boundaries of machine learning and AI applications, extending its significance far beyond the realms of gaming.

2. Methodology

2.1 Approach and Design

To provide a clear roadmap for my methodology, the approach began with the development of a baseline AI model that emulates the typical play style of an average Texas Hold'em (a type of poker game) player. This model acts as a standard against which other strategies and decision-making patterns in the game are measured. Following this, the next step involved constructing a deep reinforcement learning (RL) AI model. It would be trained to learn the nuances of poker through self-play, refining its strategies by continuously competing against itself. The culmination of this process is the selection of the most effective RL model, based on its performance. This chosen model is then trained further, this time against the baseline AI, with the purpose of developing a strategy against the baseline method of play.

2.2 Emulation Model

2.2.1 Dataset

This study's dataset is derived from the detailed hand history of "IlxxxlI," a regular player on a real money poker website. Focusing on IlxxxlI, the dataset records every hand played by this player across numerous games, regardless of the outcome. Note that only IlxxxlI's hands are shown, with the hands of players who folded or mucked remaining undisclosed. The hand histories are documented in plain English text, culminating in an extensive collection totaling 82 megabytes, which equates to well over a million lines of data. This substantial dataset offers a rich resource for analyzing IlxxxlI's decision-making strategies in the game.

2.2.2 Feature Extraction

To tune the model to mirror IlxxxlI's playstyle, it was critical to incorporate an extensive range of game state data. Given that a maximum of nine players could participate in each game, a state model was assigned to each seat. This model was designed to emulate the respective player's reaction to the moves of others at the table. To capture the actions of each opponent, I categorized their moves into five distinct actions: fold, check, call, bet, and raise. To simplify the model, the specific amounts raised were not included; instead, I focused on recording the total number of chips each player had and the amount they contributed to the pot.

Additionally, I noted each player's position at the table, categorizing them as Button, Small Blind, Big Blind, EP (Early Position), MP (Middle Position), or LP (Late Position). The status of each opponent as an active or inactive player was also tracked; players who had folded or were bankrupt were marked as inactive.

The features specific to IlxxxlI were extracted, including the number of chips, the amount contributed to the pot, their table position, the cards they held, and the actions taken by IlxxxlI.

Furthermore, the overall game state was encoded, encompassing the total chips in the pot and the community cards that were revealed.

2.2.3 Feature Encoding

After the feature extraction phase, the next step was to encode these features into tensors, making them suitable for input into the machine learning model. This encoding process involved a combination of one-hot encoding and normalization.

For each opponent, a 14-length tensor was created. The first five elements of this tensor represented the opponent's move, using one-hot encoding. For instance, if an opponent folded, the tensor would start as [1, 0, 0, 0, 0]. The table position of each opponent was encoded in a similar one-hot manner.

The amount of chips held by each opponent was normalized against the total wealth at the table. For example, if the total table wealth was $900 and the opponent had $90, this feature would be encoded as 0.1 in the tensor. The amount the opponent had in the pot was normalized relative to their total chips.

A binary value was used to denote the active status of each opponent: '0' for inactive and '1' for active.

IlxxxlI's features were encoded in a similar way. However, the action taken by IlxxxlI was recorded but not initially added to the tensor.

For encoding the cards, a 52-length tensor was used, with each position representing a different card.

The game state was encoded using techniques similar to those described above. These individual tensors were then combined into a single, long, one-dimensional tensor. At the end of this tensor, I appended the move taken by IlxxxlI. This was done to ensure that during training, the model would be able to easily slice out that data so the model wouldn't have prior access to the move it was supposed to predict.

Each round in a game was encoded as a row in a 2D tensor, with each row representing a round and having a length of 244 nodes. If an opponent's seat was empty, it was encoded as a tensor of zeros to ensure consistent length.

2.2.4 LSTM Model

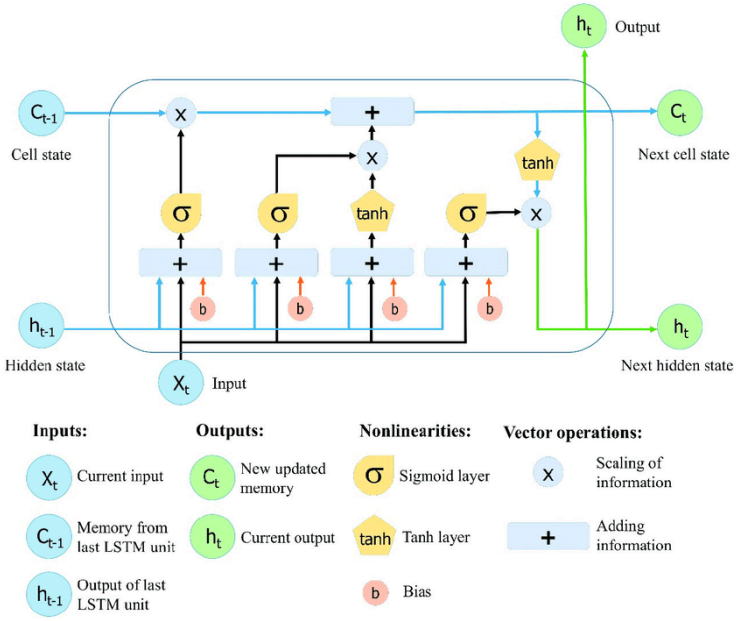


Figure 1: LSTM Model

The decision to utilize a Recurrent Neural Network (RNN) for my model was driven by the sequential nature of poker, which aligns well with the capabilities of RNNs. RNNs are uniquely suited for sequential data because they not only process the current input but also incorporate the outcome of previous network states into their predictions. This feature is particularly advantageous in a poker game environment, where each round's outcome can significantly influence the subsequent round's state.

Among various RNN architectures, I specifically chose the Long Short-Term Memory (LSTM) model. The LSTM is designed to not just consider the immediate previous output, but to take into account the entire sequence history. This capability stems from its unique architecture, which includes two types of 'memory': long-term and short-term.

The LSTM model is structured around three gates:

1. Forget Gate: This gate receives the short-term memory (also known as the hidden state) and the current input. It processes these through a fully connected neural network layer. The purpose of this gate is to determine the extent to which the long-term memory (or cell state) should be retained or modified based on the current input and recent short-term memory.

2. Input Gate: This gate similarly processes the short-term memory and the current input through a neural network. The outcome of this process contributes to the updating or alteration of the long-term memory, effectively deciding what new information should be added to it.

3. Output Gate: In this final stage, the current input, short-term memory, and long-term memory are integrated and processed through a neural network. The output of this gate serves two purposes: it becomes the next short-term memory and also forms the output of the model for the current step. Meanwhile, the long-term memory is directly carried over to the next instance without any transformation.

2.2.5 LSTM Construction

For the construction of my model, I selected specific hyperparameters to optimize its performance in the context of poker game analysis. The model architecture comprises three layers of Long Short-Term Memory (LSTM) units, a dropout layer, and a fully connected neural network layer that leads to the final output.

LSTM Layers: The core of the model consists of three LSTM layers. These layers are designed to process the sequential data, capturing the complex dependencies and patterns inherent in the gameplay. By stacking multiple LSTM layers, the model gains a deeper understanding of the sequence data, which is crucial for predicting the intricacies of poker strategies.

Dropout Layer: Positioned between the LSTM layers and the fully connected layer, the dropout layer plays a critical role in preventing overfitting. I set the dropout rate at 0.2, meaning that during training, 20% of the neuron values are randomly dropped out at each iteration. This technique helps in ensuring that the model does not become overly reliant on specific features and can generalize better to new, unseen data.

Fully Connected Neural Network Layer: The final stage of the model is a fully connected neural network layer. This layer integrates the processed information from the LSTM layers and translates it into the final output.

The output of the model is structured as a 5-length tensor, with each index corresponding to a predicted move: ["fold", "check", "call", "bet", "raise"]. This output format aligns directly with the categorization of potential moves in the game, allowing the model to provide clear and actionable predictions based on the input data and the learned patterns over the course of training.

2.2.6 LSTM Training

To effectively optimize and train my model, I employed a cross-entropy loss function in conjunction with the Adam optimization algorithm, an enhanced version of stochastic gradient descent. The choice of cross-entropy loss was strategic, as it is particularly effective when the model's output is expected to be a one-hot encoded action.

The training loop was set to run for 50 epochs. Given the immense size of the dataset, this number of epochs was a balance between thorough training and practical time constraints. Essentially, this meant the model was trained on each game in the dataset 50 times, allowing it to iteratively learn and improve its predictive accuracy.

During the training process, each game was divided into individual rounds. For the initial round of each game, I started with the long-term and short-term memory states of the LSTM network set to zeros. This initialization provided a neutral starting point for the model to begin learning the game dynamics. As the training progressed to subsequent rounds within a game, I fed the memory states from the previous round into the current one. This approach allowed the model to build upon the learned information from one round to the next, simulating the continuous flow of a poker game.

Once all rounds of a game were processed, the memory states were reset, and the model proceeded to the next game, repeating this process throughout the training loop.

2.3 Reinforcement Learning Model

2.3.1 Creating the Environment

To facilitate the reinforcement learning model, I developed a script to simulate and encode a Texas Hold'em poker game. To streamline the process and reduce complexity, I implemented two key simplifications in the game environment: the exclusion of side pots and standardizing all raises to be a fixed amount of $15.

2.3.2 Creating the Agent

The agent plays a crucial role in managing the model during the training phase of the reinforcement learning process. It is designed to perform several key functions: taking in the state of the game, determining the appropriate action, providing rewards to the models, and overseeing the training mechanism along with the management of the replay buffer.

A critical aspect of reinforcement learning is balancing exploration and exploitation. This balance is essential for the model to effectively learn the optimal strategies: exploration allows

the model to try new and varied actions, while exploitation enables it to apply the strategies that have proven successful. To manage this, I introduced an epsilon value for the agent. This epsilon-greedy strategy involves generating a random number; depending on this number and the epsilon value, the agent either selects the model's predicted move (exploitation) or opts for a random move (exploration). As training progresses, this approach ensures that the model not only capitalizes on the successful strategies it has learned but also continues to explore and adapt to new strategies and scenarios in the game. The epsilon value decays over a long period of time to ensure the model has a concrete strategy by the end of training.

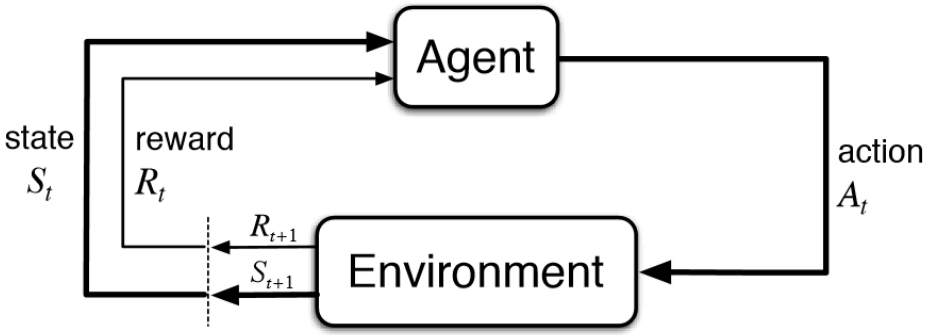


Figure 2: Agent Environment loop

2.3.3 Replay Buffer

The replay buffer in the reinforcement learning setup acts as a storage for experiences, which include the game state, the action taken by the model, the resulting state, the received reward, and a flag for the final round of a game. Its main role is to offer a diverse set of these experiences in batch samples for the AI's training.

2.3.4 Model

The type of network I would be using was a Deep Q Network (DQN), a network that rewards the agent for taking good moves, and punishes it for bad moves. The architecture of the actual model is a straightforward fully connected neural network with two hidden layers. The first hidden layer contains 256 neurons, and the second has 128 neurons. Each of these layers uses the ReLU (Rectified Linear Unit) activation function, which is effective for introducing non-linearity into the model, helping it learn complex patterns.

A notable feature of this model is the inclusion of a pseudo-attention layer. This mechanism is not a traditional attention layer as seen in models like Transformers. Instead, it serves to enhance the network's ability to focus on the most significant features or neurons during the learning process. By doing so, it allows the model to more accurately identify and prioritize crucial patterns within the data. The layer in the model was placed between the two hidden layers. It

works by first taking the output from the first hidden layer and applying a softmax function to create a probability distribution. This softmax output is then multiplied by the original output from the first hidden layer. The resulting product, which emphasizes the more significant features, is fed into the second hidden layer. This placement allows the attention layer to act as a filter, highlighting important information for the network to process more effectively. This attention mechanism is particularly useful in complex environments like poker, where discerning relevant patterns from a multitude of inputs is key to making effective decisions.

2.3.5 Deep Q Training

During training, the model processes samples drawn from the replay buffer. It begins by taking a game state from the buffer and generating a Q-value (reward) for each potential action. The model outputs a 3-element tensor, where each index corresponds to a specific move: [fold, check/call, raise/bet]. This tensor contains the predicted Q-values, which represent the expected rewards for each respective move.

In the context of maximizing rewards, the model would ideally choose the move with the highest predicted reward. However, during training, the choice is guided by the agent's decision, which might involve selecting a move different from the one with the highest immediate reward to encourage exploration.

The next step involves applying the same process to the subsequent state after the action is taken, determining the action that offers the highest reward in that future scenario. This is where the Bellman Equation comes into play. According to the Bellman Equation, the expected Q-value for a state-action pair is the sum of the immediate reward received for the current action and the discounted reward of the best action in the next state. The discount factor in this equation, typically between 0 and 1, adjusts the importance given to future rewards. If it is the last round in the game, the Bellman Equation simply returns the immediate reward of the current state.

By continuously updating its predictions based on this principle, the model learns to estimate the long-term value of actions, not just the immediate benefits, thus developing a more strategic and foresighted approach to the game.

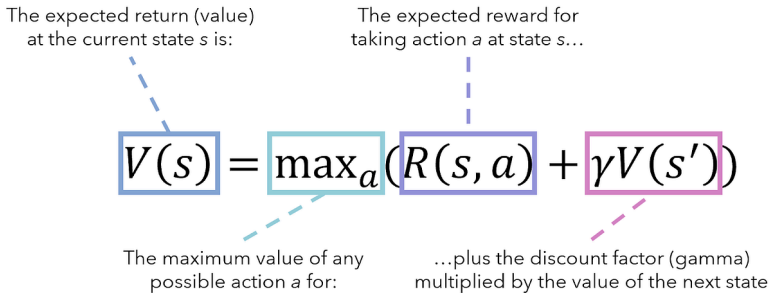


Figure 3: Bellman Equation

For calculating the loss during training, I used the mean squared error (MSE) between the model's predicted Q-values and the expected Q-values derived from the Bellman Equation. For optimization, I continued using the Adam optimizer.

2.3.6 Calculating Rewards

In the network, rewards are assigned only after the completion of a game, as the outcome of each round influences the overall result. The reward system is based on Q-values, which vary depending on the game's outcome:

1. Winning a Hand: If the bot wins, it receives a positive Q-value proportional to the amount won from the pot.

2. Losing a Hand: A negative Q-value is assigned if the bot loses, based on the amount lost.

3. Folding a Winning Hand: Folding a hand that could have won results in a negative Q-value, calculated from the potential winnings forgone and the amount already committed to the pot.

4. Folding a Losing Hand: The bot is rewarded with a positive Q-value for avoiding further losses by folding a losing hand.

5. Winning by Opponents Folding: A positive reward is given when the bot wins because all other players fold, although this reward is smaller than winning at showdown.

6. Bankruptcy: If the bot goes bankrupt, it incurs a significantly high negative Q-value.

All these values are normalized relative to the total wealth at the table, ensuring the model's adaptability to different stakes. Moreover, I scaled up all of the normalized values as I found the backpropagation did not work too well with extremely small floating point values.

After assigning these rewards, rewards are retrospectively applied to previous moves in each game. For instance, if the bot wins, earlier calls and raises are positively reinforced, but less so than the final move. Conversely, if the bot loses after raising, the action receives penalties nearly as substantial as the final negative outcome. Similarly, good folds (avoiding losses) lead to small positive Q-scores on calls and raises, while bad folds (missing potential wins) incur hefty penalties on them. This method of assigning immediate Q-scores to each action helps the bot learn effective strategies through the consequences of its decisions in each round. For those interested in seeing the exact equations used to calculate rewards, they are in the pokerutils file on github.

2.3.7 Training Loop

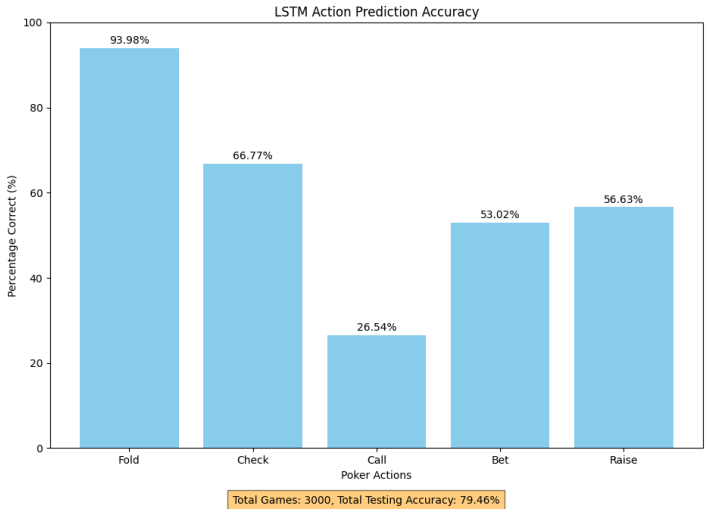
For training the models, I streamlined the process by feeding the output of a previously trained LSTM model into the state tensor. This ensured that the reinforcement learning (RL) agents started with a basic understanding of poker, rather than from scratch, accelerating their learning. I then created nine different RL agents and had them compete against each other in 25,000 games. To ensure consistent and effective training, their stack (money) was reset to the initial amount ($1000) every 15 games. This approach allowed the agents to learn and adapt in a balanced environment, enhancing their strategic capabilities in poker without the influence of long-term financial disparities.

To determine the most effective version of the bot, I conducted a comprehensive evaluation process. This involved pitting the nine RL agents against each other in a series of 50,000 games. The primary metric for success was the amount of money won, and the bot that accumulated the highest earnings was deemed the best version.

Once the top-performing bot was identified, I proceeded to the next phase of training. In this stage, the chosen bot was set to compete against eight instances of the LSTM model. Over an extensive series of 100,000 games, this rigorous training regimen was aimed at refining the model to excel against the strategy and gameplay style the LSTM held.

3. Results

3.1 LSTM Results

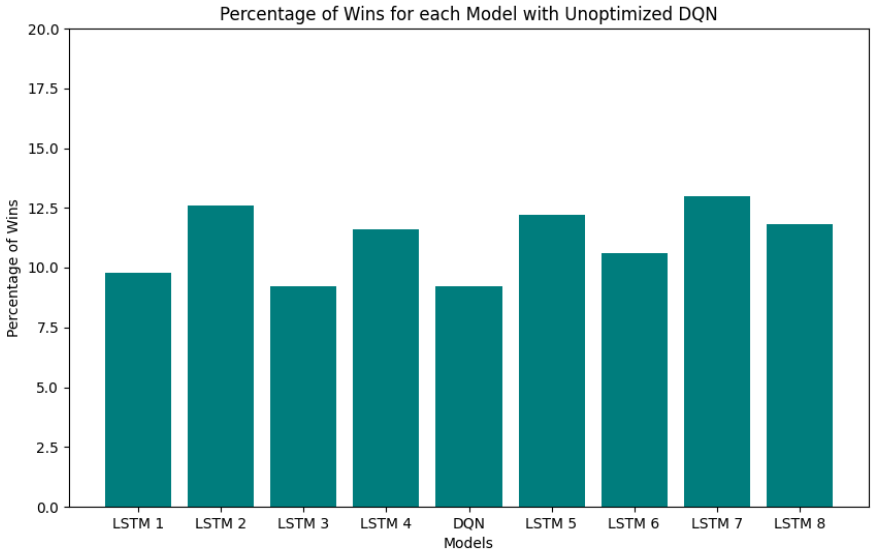


The trained model was tested on 3,000 games it had not encountered before, achieving a prediction accuracy of 79.46%. This result is particularly impressive considering poker's inherent

unpredictability and reliance on both luck and player strategy. An accuracy rate near 80% indicates that the model effectively captured a substantial aspect of the player's unique playstyle, achieving its goal.

3.2 Unoptimized DQN results

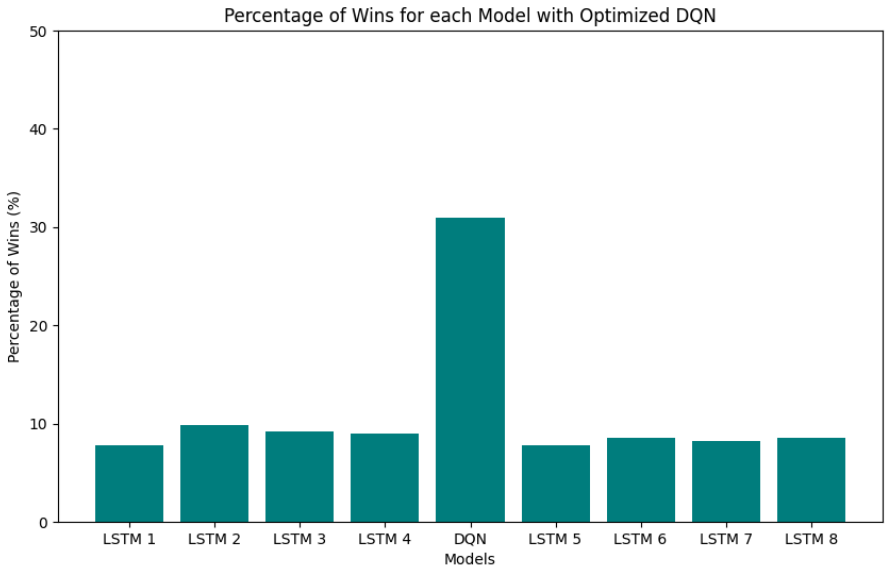
The performance of the best Deep Q-Network (DQN) model, which was trained exclusively through self-play, was underwhelming when it was tested against the LSTM models. To evaluate its effectiveness, the DQN was pitted against eight LSTM models over the course of 25,000 games. During this testing phase, every 50 games, the bot with the highest money stack was noted and given a win, and the net gain or loss in the DQN's stack was recorded. After each 50-game interval, all bots' stacks were reset to their initial value of $1000.



The DQN only won about 9.2% of the time, and earned money 17.6% of the time (since the bot can not win but still have a net positive over the 50 games). In terms of total wealth change, the DQN lost a whopping $86,053 over the course of the games.

3.3 Optimized DQN results

When the testing phase was repeated using the optimized version of the DQN, there was a significant shift in the results



The optimized DQN won 31% of the time, and earned money 50.8% of the time. The total wealth change of the DQN was a net gain of $951.

4. Discussion

In conclusion, this research provides a compelling answer to the central question: Can artificial intelligence not only master the probabilistically optimal moves, but also dynamically adapt to individual players' styles, effectively countering their unique gameplay quirks and strategies? The findings from the extensive training and testing of AI models in the complex environment of Texas Hold'em poker suggest a positive response.

The successful application of LSTM and optimized DQN models in this study illustrates AI's capacity to go beyond simply executing statistically optimal moves. These models have shown a remarkable ability to adapt to and a counter unique playstyle, reflecting significant power in AI's capability to understand and respond to human-like strategies and decision-making patterns.

This research underscores the potential of AI in decision-making scenarios that require not only a grasp of statistical probabilities but also an adaptive, strategic approach. It demonstrates prospects for AI's role in areas where understanding and mirroring human behavior are crucial, ranging from gaming to real-world applications like negotiations and behavioral predictions.

Exploring alternative reinforcement learning networks could significantly enhance this research. Two promising candidates are Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC):

1. Proximal Policy Optimization (PPO): Focused on policy learning, PPO could more directly map observed game states to optimal actions in response to a certain playstyle.

2. Soft Actor-Critic (SAC): Blending policy optimization with value function estimation, SAC could provide a comprehensive approach, balancing strategic exploitation and exploration.

Note that I tried a different actor critic model (A3C) to solve this problem, yet it yielded less success than the DQN, so I decided to omit it from the paper. For those interested the A3C code is in the github, and the model won money around 200 out of 500 games.

Comparing these methods with the existing DQN model may yield advanced AI capabilities, especially in adapting to and mastering poker's decision-making complexities, and could broaden our understanding of various reinforcement learning strategies.

5. Final Remarks

The model presented in this research, while adept at adapting to a player's playstyle in Texas Hold'em poker, is not without limitations. Particularly, it does not account for the dynamic nature of human strategy, where players may alter their playstyle in response to the AI's adaptations. To address this, a proposed enhancement involves real-time, in-game training of the AI with a high learning rate, allowing it to adjust to the player's evolving strategies. This approach, albeit experimental and requiring optimization, opens up exciting possibilities for creating more responsive and adaptive AI systems in strategic gaming environments.

Rather than detailing specific game examples played by the AI, I invite readers to experience the models in action firsthand. A fully functional poker simulator has been set up in a terminal environment, allowing for direct engagement with the AI. Users have the option to pit the models discussed in the research against each other or even challenge themselves against the AI. To partake in this experience, simply follow the instructions available in the readme file of the GitHub repository.

Code, game, and models can be found here: https://github.com/uChase/PokerBot

References

O, Oleg. “Poker Hold’Em Games.” *Kaggle*, 6 Oct. 2017, www.kaggle.com/datasets/smeilz/poker-holdem-games/data