Mastering Leduc Hold'em: A Comparative Study of Reinforcement Learning Methods

Alex L. Chandler
alex.chandler@utexas.edu
Department of Computer Science
The University of Texas at Austin

Jesse P. Judd
jpjudd@utexas.edu
Department of Electrical and Computer Engineering
The University of Texas at Austin

Abstract

Imperfect information is a hallmark of real-world strategic interactions, ranging from popular card games to business negotiations and political decision making. In this work, we apply reinforcement learning methods to competitively play Leduc Hold'em, a simplified variant of No-Limit Texas Hold'em that preserves imperfect information and betting while reducing the state-action space for computational efficiency. We model Leduc Hold'em as an episodic *Markov decision process* (MDP), where two players make iterative decisions based on partial information, receiving a monetary reward at the end of each hand. We trained 16 reinforcement learning agents in the RLCard framework and compared them to an existing CFR implementation and a highly aggressive agent named RaiseCall. Our *CFR Vanilla* and CFR^+ agents outperform all others, with CFR^+ ranking highest in both Elo and Big Blinds Per Hand (BBPH). The top-performing agents employ a balanced betting strategy capable of advanced techniques such as bluffing.[1]

1 Introduction

Artificial intelligence (AI) has made remarkable strides in solving challenging games through techniques involving reinforcement learning (RL). RL has been particularly successful in tackling two-player games of perfect information such as checkers, chess, and Go (Schaeffer et al., 2007; Silver et al., 2017). In these games of perfect information, both players have access to identical knowledge of the environment at all times, and AI systems have been shown to consistently outperform human experts.

In many games with perfect information, AI systems achieve superhuman performance by approximating Nash equilibrium strategies. A Nash equilibrium is a set of strategies, one for each player in a game, such that no player can benefit from changing their strategy while the others keep theirs unchanged. Nash equilibrium strategies, which exist for all finite games (Nash, 1950) hold additional beneficial properties for two-player zero-sum games. In two-player zero-sum games, adopting a Nash equilibrium strategy ensures a player will not lose on average, regardless of the opponent's attempts to exploit this strategy (Wimpee, 2008). Although initially developed for games of perfect information, these concepts are also applicable to games with imperfect information.[2] - the focus of this work.

When the environments complexity is evolved in such a way that each player has access to unique information, the game becomes a problem of imperfect information. Perhaps the most popular game with incomplete information is Texas Hold'em. In Texas Hold'em, each player is dealt two private cards, which are hidden from other players. The two private cards per player shape the strategic dynamics of Texas Hold'em, where players deduce opponents' hands based on betting behavior, probabilities, and psychology.

While Nash equilibrium strategies have been shown to exist in poker games (Nash, 1950), most human players do not adhere to these game-theoretic optimal (GTO) strategies (Sonawane & Chheda, 2024). Instead, human

[1] To promote reproducibility and support development, we have made all code available on GitHub. A five minute presentation summarizing this project may be found here.

[2] For the definition of imperfect information games, see the appendix and refer to Figure 8 (Zinkevich et al., 2008)

experts often play exploitatively, not only considering the cards they hold but also the cards they believe their opponent has, and even what their opponent thinks they have. As the saying goes, "You don't play the cards in front of you, you play the man across from you."[3]

Leduc Hold'em is a simplified variant of poker that maintains the imperfect information and betting of Texas Hold'em. To provide a motivation for using this simplification, Heads-up no-limit Texas hold'em (HUNL) - a two-player variant of No-limit Texas Hold'em (NLTH) - has been estimated as exceeding 10^{160} decision points (Johanson, 2013). Leduc Hold'em is an abstraction that significantly reduces the number of decision points.

Imperfect information games present several unique challenges for AI agents. In games of imperfect information, players have access to different pieces of information at a given time. The value of an action in imperfect information games depends on the probability distribution over the opponent's private information, requiring players to reason about the beliefs and strategies of their opponents. The interdependence of different stages in these games makes strategic decision-making more complex, often involving a delicate balance between exploiting the opponent's weaknesses and protecting one's own private information. Players may want to vary their strategies to avoid being exploited by opponents.

Even the simplified poker variant of Leduc Hold'em offers valuable insights into decision-making. The game incorporates imperfect information, probabilistic reasoning, risk management, opponent modeling, and deceptive tactics. In imperfect information games, the value of actions may depend on the estimated probability distribution of the opponent's private information, requiring players to reason about the beliefs and strategies of their opponents. Competitive AI approaches in imperfect information games typically reason about the entire game and produce a complete strategy prior to play, using techniques like CFR (Zinkevich et al., 2008) for recursive reasoning through self-play and iterative strategy adaptation (Gibson, 2013; Tammelin et al., 2015; Moravčík et al., 2017; Brown & Sandholm, 2019b).

2 An Overview of Relevant RL Theory

Before diving into related work, a couple of key concepts are defined. First, our ideas apply to sequential MDPs so we illustrate how to emulate such with Leduc poker. To take advantage a sequential MDP in the sense of reinforcement learning, value functions, reward functions, and policy updates are used.

2.1 Understanding the Sequentiality of Poker as a Markov Decision Process

Using traditional RL concepts, a trained agent makes informed sequential decisions to maximize cumulative future rewards - *returns* - with the environment its interacting with (Heinrich & Silver, 2016). In equation (1), the return denoted as G_t is the discounted *reward* summed across all time steps t with discount factor γ^k. The discount factor ensures that future rewards arent given too much weight since many undetermined state-action combinations will exist between the current state and future timesteps.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1}$$

For Leduc Hold'em, the agent aims to maximize the cumulative money in its pot by strategically betting on each hand. A reward, reflecting the money earned or lost, is present within each hand rollout. Players make sequential betting actions, such as raising, calling, or folding, to maximize future rewards in accordance with the game's rules. Each action taken by a player depends on the current state, which encapsulates the history of previous actions, community cards, stack sizes, and private card holdings. Ultimately, the outcome of each hand is determined by the combination of the agents' actions and the probabilities associated with the dealt cards.

[3] A variant of the phrase was said by James Bond in the film "Casino Royale". However, the expression is believed to predate the character's use.

2.2 Trajectories: Influencing Sequential MDP's

To provide better influence and learning to an agent, *trajectories* are commonly utilized instead of singular state-to-state transitions. In the context of Leduc Hold'em, a trajectory array would contain entries of state-action-rewards tuples for each action made until the hand ends. The trajectory provides the agent with more recollection of past event details with the goal of the agent making better-informed decisions. The tree diagram in Figure 1 for Leduc Hold'em illustrates some possible trajectories that will require exploration.

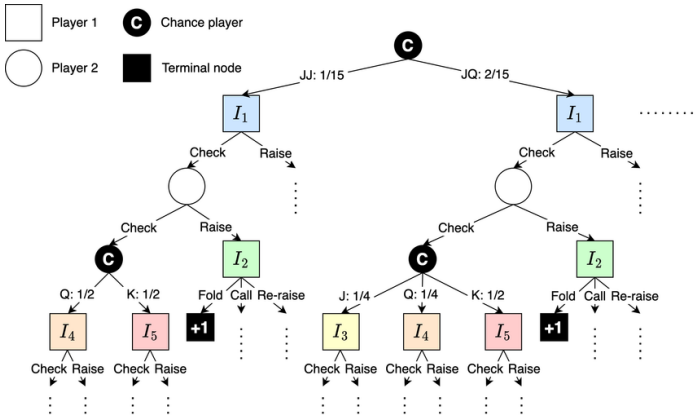

Figure 1: Game Tree for Leduc Hold'em with Possible Trajectories (Habara et al., 2023)

The actual trajectories experienced will vary by agent depending on their value functions and policies.

2.3 Value Functions and Policy Updates

Each agent we trained learns a modified value function and/or policy update. The value function assesses the long-term reward from any given state, guiding decisions at each step by estimating future benefits. In contrast, the action-value function evaluates the expected reward for each state-action pair, as formalized in Equation (2) (Sutton & Barto, 1998).

$$Q(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2}$$

Following a policy π, the action-value function is the expectation of getting a return by taking action a from state s. The policy defines how an agent will behave or *act* given its experience and current state (3).

$$\pi'(s) = \underset{a}{argmax}(Q_\pi(s,a)) \tag{3}$$

Our agents, listed in Figure 2, differ in their use and update of value functions (state-value or action-value) and policy functions, with some methods relying on state-value functions $V(s)$ instead of $Q(s,a)$ for policy updates. These differences in the underlying reinforcement learning algorithms lead to distinct agent behaviors and performance in the Leduc Hold'em environment. We discuss our agents in Section 4.[4]

[4]Further information of some of our proposed variants may be found in the Appendix.

CFR	Sarsa	Other
CFR RLCard	Sarsa	Raise Call
CFR Vanilla	Expected Sarsa	Random
CFR+	Sarsa(λ)	Q-Learning
Deep CFR	Sarsa(λ) Variant	REINFORCE
MCCFR-Outcome	N-step Sarsa	
MCCFR-External	N-step Sarsa Variant	
MCCFR-Average	True Online Sarsa(λ)	
	Deep True Online Sarsa (λ)	

Figure 2: Table of Agents

3 Understanding Leduc' Holdem

Leduc Hold'em is a two-player poker game containing a deck of six cards, with two of each of the following face cards: Jack, Queen, and King. Each hand requires a shuffle prior to dealing players their cards (Hawkin et al., 2011). The players start with an equal amount of funds and take turns being the dealer, with the non-dealer placing a small blind bet and the dealer placing a big blind bet. Each hand, from beginning to end, is treated as one episode.

In Leduc Hold'em[5], each player is dealt one private card, and a single shared community card, known as the *flop*, is dealt between the two rounds of betting. During their turn, a player selects from the legal actions available, which is a subset of the following: (1) to *raise* the bet by adding to the pot, up to a maximum amount allowed in each round; (2) to *call* the current bet, matching the amount added to the pot by the opponent; (3) to *check*, passing the action to the opponent without adding to the pot (only available if no bet has been made in the current round); or (4) to *fold*, ending the hand and allowing the opponent to win the pot. If a player raises and the opponent does not have enough funds to call, the opponent must either fold or go all-in with their remaining funds (Waugh et al., 2009).

After both players have taken one action, the community card is publicly revealed by the dealer. Each player has new information about what they perceive their opponent has and if they have a match to the flop. Another round of betting ensues following similar rules of the first round. If both players have acted and neither has folded by the end of the second betting round, a showdown occurs. At showdown, the winner is determined as the player with the equivalent card to the community card (ie the community card is a Q and the winner has a Q). If neither player has the community card, the player with the highest face value card wins the pot (ie winner has a Q, loser has a J, and community card is K). If both players have the same card, the money in the pot is evenly split.

The game ends when a player runs out of money or after a set number of hands, with the winner having the most money. The players then switch positions, with the small blind becoming the big blind and vice versa, and a new hand begins with a shuffled deck.

4 Related Works and Literature

4.1 A Tractable Environment

To handle the massive computational requirements of solving both NLTH and HUNL, researchers have employed abstraction techniques, such as state abstraction and action abstraction.

State abstraction involves consolidating similar game states into uniform information sets, often by clustering hand combinations based on their expected hand strength (Johanson, 2007). Action abstraction simplifies a player's choices by dividing the action space into bins, such as betting in fixed fractions relative to the pot

[5]Leduc Hold'em differs from traditional Texas Hold'em, where the flop consists of three cards, and a total of five community cards are revealed rather than just one.

size (e.g., quarter pot, third pot, half pot, full pot, overbet) or choosing non-bet actions like check or fold. Both of these abstractions significantly reduce the complexity of the game tree.

By employing abstraction techniques in HUNL, researchers have simplified the game to approximately 10^{12} decision points while still achieving superhuman performance (Zinkevich et al., 2008; Bowling et al., 2015).

While these direct abstraction techniques expedite training and provide additional performance for large state-action space games, simpler versions of these two-player games like Leduc Hold'em and Kuhn Poker (Kuhn, 1950) have been preferred by researchers because they still maintain the key elements required to gain insight on hidden information game-solving techniques (Shi & Littman, 2001) without requiring direct abstraction. Instead of the former, these simplified variants like Leduc Hold'em and Kuhn Poker *are* simplified abstractions of Texas Hold'em. These simplified variants allow researchers' development and testing of new algorithms and strategies while reducing the computational burden. This insight holds the intention of being beneficial to larger games such as HUNL that do directly require state and action abstractions to be efficiently tackled.

4.2 Reinforcement Learning Methods

4.2.1 Monte Carlo Learning

Monte Carlo (MC) methods for learning value functions estimate the value of states by averaging the returns observed from those states through sampling trajectories. Value estimates and policies are updated at the end of each episode (Sutton & Barto, 1998) - or in the case of poker, each hand. Monte Carlo methods can operate in both online and offline modes. Offline learning allows for the use of pre-existing behavior policies and does not necessitate the agent's active interaction within the environment, making it suitable for scenarios where the entire dataset is available from the start. In contrast, online learning involves continuous updates to the model with each new piece of data, requiring the agent to act and adapt within the environment in real-time,

4.2.2 Q-Learning

Q-Learning is a popular and foundational off-policy method that utilizes a greedy policy, selecting the action with highest estimated Q-Value. Q-Learning is an incremental method for dynamic programming and does not require a model (Watkins & Dayan, 1992). Evaluations of action-values are generally updated iteratively untill convergence.

If the experiences consist of episodes as in Leduc Hold'em, the agent selects the best estimated action a_t^* from state s_t. Upon moving to the new state s_{t+1} due to action a_t^*, it receives an immediate reward r_{t+1} and updates its previous Q-Value $Q(s_t, a_t)$ to a new estimate Q_t. A basic form of the Q-Learning update, as seen in (Sutton & Barto, 1998), is given by Equation (4).

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4}$$

One caveat of Q-Learning is that we need to provide intelligently procured initial Q-Values to facilitate good learning, especially at the infancy stages of training (Watkins, 1989).

4.2.3 Sarsa Learning

Sarsa is a popular learning method that utilizes temporal difference (TD) control methods Sutton & Barto (1998). TD control considers the current state and action as well as a temporally displaced state and action as a simplified type of trajectory. Similarly to Q-Learning, Q-Values need to be initialized.

The trajectory of Sarsa methods is self explanatory s, a, r, s', a'. An action a is taken from state s using an ϵ-greedy policy for existing Q-Values. The action moves the agent to state s', receiving an immediate reward r. Before Q-Values are updated (5) with the new action-values based on the observed reward r, another action a' is chosen (Sutton & Barto, 1998). The state and action, s and a, are updated with the new state and action,

s' and a', before moving to the next step of the episode. If the subsequent state is terminal $S' = S^T$, the associated Q-Value for $Q(S', A')$ is fixed to 0.

$$Q(S, A) = Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)] \tag{5}$$

Sarsa learning methods employ a learning rate α and a discounting factor γ as important hyperparameters required for optimization. Thorough characterization needs to be considered when developing optimal Sarsa algorithms.

4.2.4 Counter-Factual Regret Learning

CFR (Zinkevich et al., 2008) has been the leading framework for solving large state space imperfect-information games. CFR is an iterative self-play algorithm that minimizes *regret*, a concept from decision theory representing how much a player regrets not having chosen a different action in hindsight (Savage, 1951). The CFR algorithm is proven to converge to a Nash equilibrium in two-player zero-sum games (Zinkevich et al., 2008). A key concept in CFR is *counterfactual value*, which represents the expected value of reaching a state if a player tries to reach it. CFR uses these values to compute regrets for each action and update its strategy.

4.2.5 Counter-Factual Regret Plus (CFR$^+$) Learning

CFR$^+$ is a popular CFR variant that prevents the indefinite accumulation of negative regrets by zeroing out negative regret values, enabling faster convergence to Nash equilibria (Tammelin, 2014). CFR$^+$ achieves higher efficiency and quicker convergence compared to traditional CFR, often called Vanilla CFR, in large-scale games. (Tammelin, 2014).

4.2.6 Linear Counter-Factual Regret (Linear CFR) Learning

To further improve convergence, variants like *Linear CFR* can be utilized in the early stages of training. Linear CFR improves upon Vanilla CFR by adding counterfactual deltas to the cumulative regret metric at each timestep rather than replacing them, increasing the speed of Nash equilibria convergence (Brown & Sandholm, 2019a).

Linear CFR is also advantageous as it employs a more aggressive decay rate at the early stages to reduce the influence of random starting strategies that are probable to be far from the optimal. The CFR methodology instills what can be thought of as a form of memory, allowing for iterative improvements that hinder learning at the infancy stages of training. As self-play progresses, the agent will still converge towards the Nash equilibrium, with Linear CFR accelerating this process.

4.2.7 Monte Carlo Counter-Factual Regret (MCCFR) Learning

Monte Carlo CFR (Lanctot et al., 2009) samples actions in the game tree to handle larger games while recent variants like Discounted CFR (Brown & Sandholm, 2019a), Variance Reduction Monte Carlo Counterfactual Regret (VR-MCCFR) (Schmid et al., 2018), Pure Monte Carlo Counterfactual Regret Minimization (Qi et al.) have further improved CFR's scalability and performance by losing the requirement to update regret values across all decision points in the game tree.

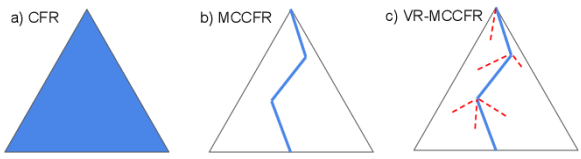

Figure 3: Tree Traversals for three CFR variants (Schmid et al., 2018)

4.2.8 Deep Counter-Factual Regret (Deep CFR) Learning

Breakthrough poker AIs like Libratus (Brown & Sandholm, 2018), Pluribus (Brown & Sandholm, 2019b), and DeepStack (Moravčík et al., 2017) have used CFR-based methods to defeat humans professionals in heads-up and multiplayer no-limit Texas Hold'em. Libratus and Pluribus compute strategies offline using CFR and detailed abstractions. DeepStack instead uses depth-limited lookahead and learned value functions to estimate values at the depth limit, avoiding full strategy precomputation and abstraction. This allows real-time strategy computation for arbitrary situations. Deep CFR, introduced by (Brown et al., 2019), uses deep learning to directly approximate CFR without abstraction. It is the first non-tabular CFR variant to achieve strong performance in large poker games. Neural Fictitious Self Play (NFSP) (Heinrich & Silver, 2016) has also applied deep learning to imperfect-information games, but using fictitious play rather than CFR. However, NFSP converges slower than CFR variants in practice (Brown et al., 2019; Chen et al., 2023)

4.2.9 Large Language Model's (LLM) for Poker

Recent work has also explored adapting Large Language Models (LLMs) to play poker. PokerGPT (Huang et al., 2024) fine-tunes a pre-trained LLM on textual game records to generate poker decisions. While intriguing, current LLMs do not appear to play Game-Theoretic Optimal (GTO) poker (Gupta, 2023), and existing evaluation for LLM based poker solvers does not compare to SOTA models. DecisionHoldem (Zhou et al., 2022) uses depth-limited solving considering opponent hand ranges to reduce strategy exploitability. Alternative techniques like evolutionary learning have also been applied to evolve poker strategies through co-evolution (Thompson et al., 2008). Recent work has applied actor-critic methods to learn optimal policies in multiplayer poker (Srinivasan et al., 2018; Shi et al., 2022). These approaches use an actor network to make decisions with imperfect information and a critic network to evaluate policies with perfect information.

4.2.10 REINFORCE Learning

REINFORCE is a policy gradient method that estimates the gradient of the expected return with respect to the policy parameters, without explicitly computing the gradient (Williams, 1992).

In Equation (6), the policies parameter vector is updated and the second terms expectation when following π is exactly that of the stochastic gradient descent (Sutton & Barto, 1998):

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)} \tag{6}$$

REINFORCE always uses the complete return available at timestep t. In this sense, REINFORCE is similar to Monte Carlo methods discussed.

4.2.11 Expected Sarsa Learning

Expected Sarsa learning (8) is almost identical to Q-Learning. Instead of taking the max of Q as done in Equation (4), Equation (7) uses the expected value (Sutton & Barto, 1998). The expected value is more of a likelihood of a policy taking an action a from a state s.

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi [Q(S_{t+1}, A_{t+1})|S_{t+1}] - Q(S_t, A_t)\right] \tag{7}$$

$$= Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{8}$$

Expected Sarsa is deterministic and typically yields better performance over Sarsa but is more complex to implement.

4.2.12 n-step Sarsa Control

For n-step Sarsa control methods, the return is computed as the discounted rewards from looking n-steps into the future, taking actions as 4.2.3 does. For n-step returns, the *update target* is calculated from estimated action values (9).

$$G_{t:t+n} = \gamma^0 R_{t+1} + \gamma^1 R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}), n \geq 1, 0 \leq t < T - n \quad (9)$$

When the trajectory goes beyond the terminal state s^T or ends at the terminal state, (9) simplifies to $G_{t:t+n} = G_t$ and the update is given by (10):

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)], 0 \leq t < T \quad (10)$$

n-step methods aim to improve over there 1-step abstractions by gleaning more insight into future state-action-reward tuples. Figure 4 shows the backup diagrams for several variations:

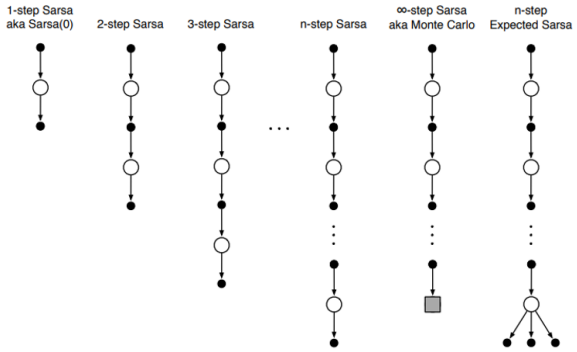

Figure 4: State-action backup diagrams for various n-step methods (Sutton & Barto, 1998)

4.2.13 Sarsa(λ) Learning

To implement Sarsa(λ), the concept of eligibility traces[6] are utilized. This forward view approach involves the nuance of updating and storing several weight and trace vectors which is generally computationally unfavorable.

The update target for these methods is a TD error updated on each step of the episode. The TD error is used to modify the weight vector $\boldsymbol{w}$. And subsequently, the Q-Value is updated based on the next state S' and the weight vector $\boldsymbol{w}$ and an action is selected (Sutton & Barto, 1998).

4.2.14 True Online Sarsa(λ) Learning

In true online Sarsa(λ), updates to the feature vector are functions of the state s and action a, and hence are available at each step. Updates to action values can occur more frequently aiding faster learning than in equivalent offline methods. Computational overhead is exacerbated by the frequency of updates performed in true online methods.

5 Methodology

We provide an overview of the RLCard Environment, the action and state space representations, the data used to train agents, our reward function, metrics, and baselines for agent evaluation. See figure 2 for all agents used and trained in this study.

[6]Eligibility trace definitions are beyond the scope of this work. They are defined in thorough detail by Sutton & Barto (1998).

5.1 Environment Overview

Our work leverages existing card game environments implemented by *datamllab* in the RLCard toolkit (Zha et al., 2020a). The RLCard toolkit enables us to train and interact with our agents. The Leduc Hold'em' environment includes a graphical user interface (GUI) that can be utilized through an IDE. This rudimentary GUI provides easy access to debugging implementations.

Once training is done, models can then be imported into a RLCard derivative: *RLCard-Showdown* (Zha et al., 2020b). RLCard-Showdown supports a web-based GUI built off of React, Django, and Flask. The setup and debug for this GUI is cumbersome. Modifications or debugging via RLCard-Showdown are simply inefficient. The GUI is used after training to provide visual playback for simulated runs between agents selected for comparison. This includes the capability to upload independently trained models, compare payoffs, and review tournament playback.

5.2 Data Representation

5.2.1 Action Space Representation

The action space representation follows quite naturally from the abstractions introduced in the transformation of NLTH to Leduc Hold'em. There exist four actions in Leduc Hold'em, where at any given time, a maximum of three of these four actions are available for each player. For example, it does not make sense to fold when you can check, and one can only raise if they have the remaining chips to do so. Our discretized action space is represented by four binary values, each representing one of the actions seen below.

Action Representation: A 1D Binary Array of One-Hot Encoded Values
Action $\leftarrow$ (Call $\in 0, 1$, Raise $\in 0, 1$, Fold $\in 0, 1$, Check $\in 0, 1$)

5.2.2 State Space Representation

We adopt RLCard's compatible state space representation for the game of Leduc Hold'em, which is a 1D array of 36 binary values representing the private cards, community cards, and each player's chip count. However, this state representation is action agnostic, a decision we view as a major limitation of the RLCard library. Ideally, the state space should include betting history, which is necessary for predicting the opponent's cards and enabling exploitative play. In exploitative play, agents identify and take advantage of their opponents' mistakes and tendencies, even though the Nash equilibrium strategy is inexploitable by opponent behavior in Leduc Hold'em (Wimpee, 2008).

State Representation: A 1D Binary Array of One-Hot Encoded Values
State $\leftarrow$ (Your Card $\in \{J, Q, K\}$,
Public Card $\in \emptyset$ if in first round, $\in \{J, Q, K\}$ in second round
Your Chip Count $\in \{0, 1, \ldots, 14\}$,
Opponent's Chip Count $\in \{0, 1, \ldots, 14\}$)

5.2.3 Dataset

For each agent, except Q-Learning, we generate a dataset of game outcomes through iterative self-play, a technique that has shown success in solving imperfect information games such as poker (Brown & Sandholm, 2018). The dataset changes for the training of each bot as it learns from its own gameplay. For Q-Learning, we use a history of hands from prior agents to facilitate off-policy learning.

5.3 Reward Function

We define our reward function as the number of chips lost or gained at the end of the hand, also known as the payoff. This payoff is calculated by dividing the net chips gained per hand by the big blind, resulting in the Big Blind Per Hand (BBPH) metric. The `RLCard` environment calculates the payoff as follows:

$$\text{payoff} = \frac{\text{net\_gained\_chips}}{\text{big\_blind}},$$

which is equivalent to BBPH.

5.4 Metrics

We employ two metrics to evaluate the performance of our poker AI: Elo rating and Big Blinds Per Hand (BBPH). Elo, originally designed for ranking chess players, has been adapted to measure skill in various games, including poker (Lambrecht, 2020). It is particularly useful when comparing a large set of opponents playing heads-up, where not all players face each other simultaneously. Unlike BBPH, Elo takes into account the strength of the opponent, assigning more weight to wins against top-performing players compared to weaker opponents. We use the Elopy library to calculate Elo ratings across all games and matchups (Barlowe, 2021). In our Elo calculations, we define a game as 100 continuous hands, which we believe is a reasonable estimate of the average number of hands played in a single poker session, although this number can vary widely from person to person. Each agent plays 80 games heads-up against every opponent[7]. We rank opponents by Elo and calculate the margin of error for each Elo calculation.

BBPH, our second metric, is equivalent to one-thousandth of the common metric milli-big blinds per game (mbb/g) (Moravčík et al., 2017; Brown & Sandholm, 2019b). We prefer BBPH because it aligns with how Texas Hold'em players typically think in terms of big blinds, and it matches our reward system, which is the payoff of net chips gained per big blind, or big blinds per hand. Additionally, we find the language clearer when using "per hand" since mbb/g actually measures the average milli-big blinds per hand, not per game. As mentioned earlier, in our Elo calculations, we define one game as 100 hands.

5.5 Baseline Comparison Strategy

We create the following baselines to evaluate our trained agents.

Baseline on existing bots for the game of Leduc Hold'em

- Create bots that do the following:
  - Perpetual Raise/Call (Raises if raising is an option, else calls)
  - Perpetual Random
- CFR RLCard (we train RLCard's existing CFR implementation)

[7]The RLCard benchmark evaluation code was quite slow, but more games would give us Elo rankings with lower margins of error.

6 Results and Evaluation

6.1 Comparative Results

Agent	Action Percentage				Elo
	Call	Raise	Fold	Check	
CFR$^+$	26.9	25.8	20.8	26.5	1531.2 ± 20.3
CFR Vanilla	26.8	23.9	19.8	29.5	1501.8 ± 20.3
CFR Vanilla (Theirs)	27.6	22.7	19.0	30.7	1499.8 ± 20.3
Sarsa(λ) Variant	39.7	23.4	18.5	18.5	1382.3 ± 20.5
MCCFR-Avg Strategy	25.3	24.7	25.2	24.7	1356.0 ± 20.6
MCCFR-External	26.9	26.7	20.3	26.2	1337.5 ± 20.3
True Online Sarsa(λ)	27.2	28.6	19.6	24.7	1305.0 ± 20.3
Sarsa	24.9	32.7	20.6	21.8	1302.3 ± 20.4
Q-Learning	23.8	36.7	19.4	20.1	1286.3 ± 20.3
Expected Sarsa	23.8	36.3	20.1	19.8	1286.2 ± 20.4
Deep True Online Sarsa(λ)	23.9	39.0	18.5	18.5	1286.0 ± 20.5
N-Step Sarsa Variant	26.1	31.2	19.9	22.7	1280.2 ± 20.2
MCCFR-Outcome	25.9	26.6	21.1	26.3	1284.6 ± 20.3
Deep CFR	25.9	24.6	24.9	24.6	1285.0 ± 20.4
Sarsa(λ)	23.9	39.0	18.5	18.5	1274.3 ± 20.5
Raise Call	27.4	62.7	0.0	0.0	1271.1 ± 20.5
REINFORCE	25.4	25.9	24.6	24.0	1259.4 ± 20.2
N-Step Sarsa	24.3	33.2	22.8	19.7	1250.6 ± 20.3
Random	25.3	24.8	25.2	24.7	1247.6 ± 20.5

Table 1: Average percentages across agents, where each agent played 8,000 hands of poker against each other. Elo numbers are shown with 95% confidence intervals. We see that the top performing agents utilize a balanced betting strategy.

Table 1 shows the Elo rankings of all agents, along with their action percentages. We see, as demonstrated by the Elo rankings in Table 1, that 14 of our trained agents outperform our best performing Naive Agent. We find that the ultra-agressive RaiseCall bot outperforms two of our trained agents and that the Random agent has the worse performance. Two of our agents, CFR$^+$ and CFR Vanilla outperform the existing RLCard CFR Implementation. We find that all top-performing bots employ a balanced betting strategy, with folding as the least common action in eight of our top 10 bots and all three of our top performing bots.

Figure 5 shows a 2D matrix with the pairwise BBPH of each opponent playing each other, sorted by the overall average BBPH across all games. Figure 5 represents the BBPH over the same evalaution run as seen in Figure 1. To our surprise, we find significant differences in ranking between Average BBPH of an agent and its calculated Elo. We believe that the difference in ranking between these two metrics is due to Elo factoring in the strength of the opponent, along with the need of more evalaution runs. The biggest seen difference is that the Raise Call agent is ranked a lot lower when using Elo over average BBPH. When analyzing the pairwise performance of RaiseCall against other agents, it becomes clear that RaiseCall performs poorly against the majority of agents, but has stellar performance against a select few of the poorly performing agents such as REINFORCE and Random bots. We find the Elo ranking more appropriate, as high rewards over poor performing bots skews the ranking of agents when their performance should more heavily factor the performance against top players.

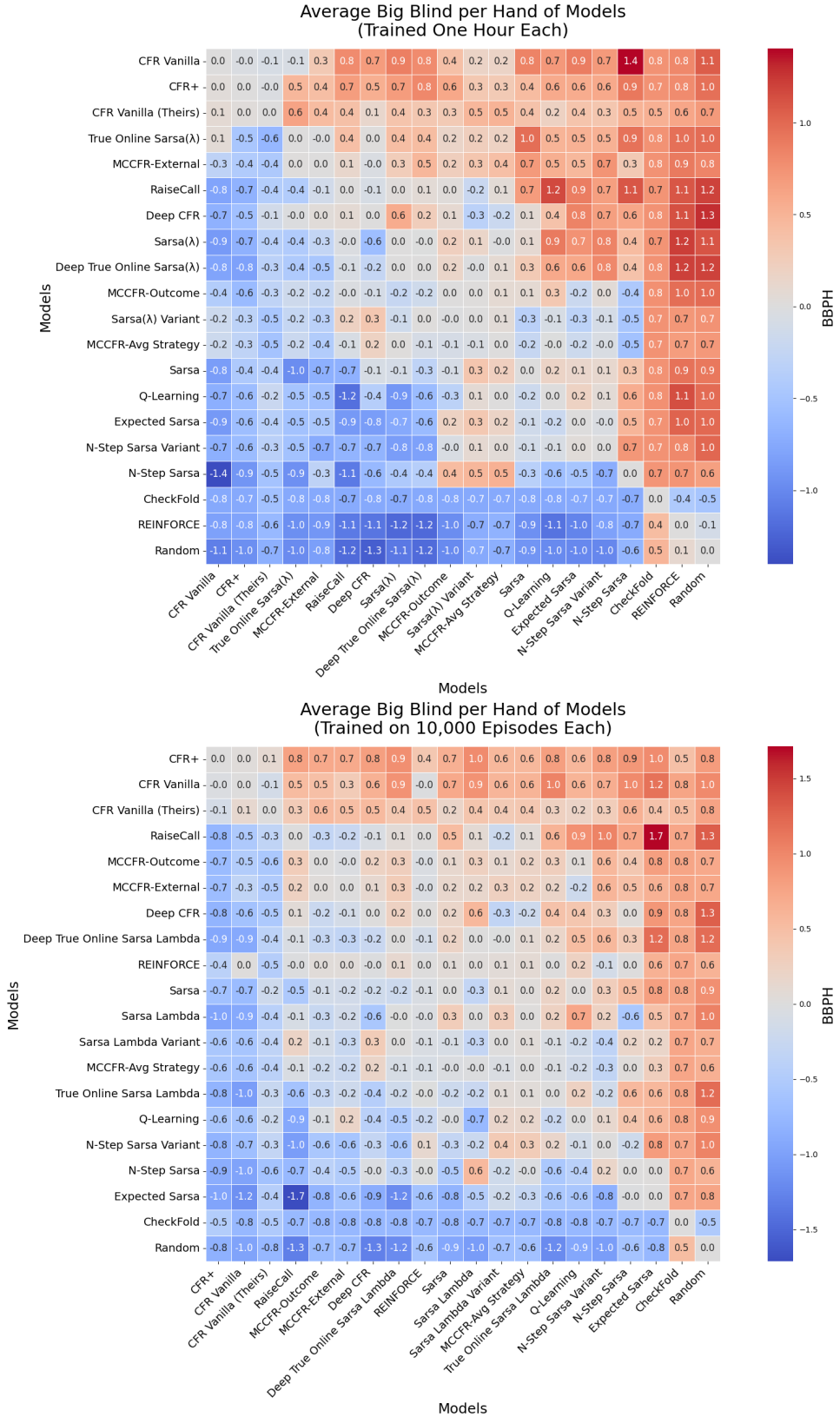

Figure 5: Top: A matrix comparing poker AI model performance in head-to-head play. Values show the net average big blinds won/lost per hand (BBPH) over 8,000 hands. In this figure, each model was trained for one hour. Dark red indicates best performance. Bottom: Each model was trained on 10,000 hands.

6.2 Analysis

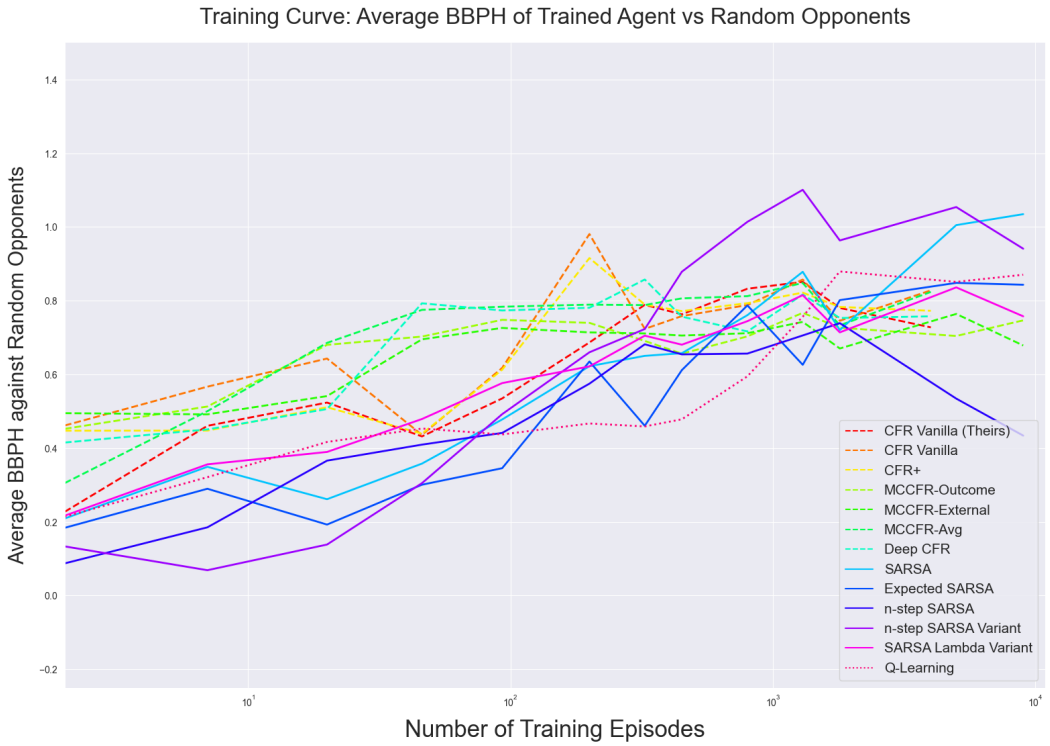

Figure 6: A plot of the reward (measured by BBPH) at training checkpoints where the agent plays against a random agent over 1,000 hands. This figure is used to visualize learning instability rather than as a performance measure, as evaluating against a random agent is a weak indicator of agent performance.

CFR methods excel in Leduc Hold'em, as they have been theoretically shown to approximate the Nash Equilibrium in zero-sum, two-player games. In such games, any Nash Equilibrium solution is inexploitable, regardless of the opponent's behavior. Other agents, which learn through self-play, likely struggle against ultra-aggressive strategies like RaiseCall because they are trained against themselves rather than trajectories of previous hands that include such extreme behavior as RaiseCall.

Figure 7 demonstrates our CFR agents' capability to vary its strategy and bluff against opponents. After the flop, the CFR agent holding the Jack of Spades continues to bluff against its opponent with the Queen of Spades. By betting 4 chips into an 8-chip pot, the agent requires the opponent to fold at least one-third of the time for the bluff to be profitable. This requirement is quantified using "bluff equity," the percentage of successful bluffs needed for profitability, calculated as:

$$\text{Bluff Equity} = \frac{\text{Amount Risked by Bluffer}}{\text{Total Pot After Bluff}} = \frac{4}{12} = \frac{1}{3} \approx 33.33\%$$

Moreover, if a King appears as a community card, only one King remains in the deck, decreasing the probability of an opponent having a strong enough hand to call the bet without a pair in a substantial pot.

Figure 7: In this figure, we see our trained CFR agent, seen in the top of the figure, bluffing in the second round of betting with the worst possible hand.

7 Limitation and Future Work

Based on two-sample t-tests, CFR$^+$ outperforms all other agents in our study. However, we are cautious to provide a definitive ranking of all agents due to the following limitations: (1) The high margin of error resulting from limited computational resources and non-optimized RLCard evaluation code. (2) The reward and performance of each agent varies significantly across training runs and episode checkpoints, as shown in Figure 6, with significant instability observed particularly for non-CFR methods. (3) We did not perform hyperparameter optimization across all agents due to time constraints, which could have significantly changed the comparative performance of each agent. The significant variance in training reward per epoch also highlights the need for such optimization (4) We acknowledge the possibility of bugs or issues in our agent implementations, which may have affected their performance. (5) As discussed in Subsection 5.2.2, RLCard environment's state representation is overly simplistic and does not properly include the entire action history, a major shortcoming that prevents the agents from learning exploitative poker strategies. In particular, agents that utilize function approximation could have potentially benefited from a larger state space that includes betting history.

Leduc Hold'em itself is an oversimplistic representation of Texas Hold'em, reducing the needed abstractions for real poker. We also found significant variation in performance between evaluation runs for SARSA agents, Q-Learning, and REINFORCE, emphasizing the need for more robust statistical tests and calculation of average Elo and BBPH across different training runs. Many non-CFR agents lose heads-up to the ultra-aggressive RaiseCall agent, possibly due to learning solely through self-play, the action agnostic state-representations, and perhaps thus insufficient training data to counter exploitative strategies. Deep methods did not perform as well, likely because they used a simple state space, with their benefits being more evident in games with larger state and action spaces.

Future work could involve more in-depth action analysis, such as examining the types of hands on which top-performing bots bluff and comparing action percentages for each round of betting between the small and big blind positions. Future off-policy learning for poker variants should utilize recorded hands from human

play or diverse strategies from various learned agents. This project was extensive, but in many ways, it feels like we have only scratched the surface. I plan on rewriting the RLCard repository's Leduc Hold'em section to incorporate betting history into the state representation. Extending these agents to NLTH would require significant and non-trivial state and action abstraction. In addition we wish to apply and compare more actor-critic methods like Proximal Policy Optimization (Schulman et al., 2017) and Soft Actor-Critic (Haarnoja et al., 2018) on No Limit Texas Holdem while experimenting with a continuous action space.

8 Conclusion

In this study, we treat Leduc Hold'em as a reinforcement learning problem. We implement, train, and test various Counterfactual Regret Minimization (CFR) and reinforcement learning methods to competitively play the game. By evaluating these methods against each other, we demonstrate that CFR methods consistently outperform the other reinforcement learning approaches. Two of our agents, CFR$^+$ and CFR Vanilla, outperform all other trained agents in the game of Leduc Hold'em. Our analysis reveals that the top-performing agents utilize a balanced betting strategy, with folding as the least common action. Furthermore, we discover that our top-performing trained bots are capable of employing varying strategies such as bluffing.

Broader Impact Statement

While reinforcement learning in poker provides insights into human psychology, education, and security, our main ethical consideration is that publicly providing solved agents for the game of No-Limit Texas Hold'em (NLTH) may enable cheating. Fortunately, our work on Leduc Hold'em is not directly applicable to this problem, as Leduc Hold'em is primarily played within the academic community, and transferring our CFR and RL implementations to NLTH would be a non-trivial task. As these techniques become more accessible for NLTH, caution should be exercised when sharing code or complete solutions. The deployment of RL agents in security-sensitive contexts raises concerns about potential algorithmic biases and unintended consequences. It is crucial to balance the benefits of integrating AI technologies into poker and other domains with measures that mitigate negative impacts and safeguard the well-being of individuals and communities.

Acknowledgments

We owe our gratitude to Dr. Peter Stone[8] and Dr. Amy Zhang[9] for enabling our Reinforcement Learning investigation. And to Michael Munje[10] and Shuozhe Li[11] for feedback and guidance through out our research.

References

Matt Barlowe. elopy: A package to simplify elo calculations. https://github.com/mcbarlowe/elopy, 2021. Python package version 0.1.4.

Michael Bowling, Neil Burch, Michael Johanson, and Oskari Tammelin. Heads-up limit hold'em poker is solved. *Science*, 347(6218):145–149, 2015. doi: 10.1126/science.1259433. URL https://www.science.org/doi/abs/10.1126/science.1259433.

Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

Noam Brown and Tuomas Sandholm. Solving imperfect-information games via discounted regret minimization, 2019a.

Noam Brown and Tuomas Sandholm. Superhuman ai for multiplayer poker. *Science*, 365(6456):885–890, 2019b.

[8] ACM Fellow, co-founder of Cogitai, Inc and Executive Director of Sony AI America

[9] PhD at McGill University and Mila - Quebec AI Institute and M.Eng. in EECS and dual B.Sci. degrees in Mathematics and EECS from MIT

[10] PhD student at UT Austin, M.S. in Computer Science at Georgia Tech with a specialization in Machine Learning, and B.S. in Computer Science at California State University Northridge

[11] Department of Computer Science at The University of Texas at Austin

Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization, 2019.

Jiayu Chen, Tian Lan, and Vaneet Aggarwal. Hierarchical deep counterfactual regret minimization. *arXiv preprint arXiv:2305.17327*, 2023.

Richard Gibson. Regret minimization in non-zero-sum games with applications to building champion multiplayer computer poker agents, 2013.

Akshat Gupta. Are chatgpt and gpt-4 good poker players?–a pre-flop analysis. *arXiv preprint arXiv:2308.12466*, 2023.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

Keigo Habara, Ellen Fukuda, and Nobuo Yamashita. Convergence analysis and acceleration of the smoothing methods for solving extensive-form games, 03 2023.

John Hawkin, Robert Holte, and Duane Szafron. Automated action abstraction of imperfect information extensive-form games. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25:681–687, Aug. 2011. doi: 10.1609/aaai.v25i1.7880. URL https://ojs.aaai.org/index.php/AAAI/article/view/7880.

Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games, 2016.

Chenghao Huang, Yanbo Cao, Yinlong Wen, Tao Zhou, and Yanru Zhang. Pokergpt: An end-to-end lightweight solver for multi-player texas hold'em via large language model, 2024.

Michael Johanson. Measuring the size of large no-limit poker games, 2013.

Michael Bradley Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player. 2007.

H W Kuhn. A simplified two-person poker. *Contributions to the Theory of Games*, 1:97–103, 1950.

Marco Lambrecht. Measuring skill and chance in different versions of poker. 0687, 08 2020.

Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. pp. 1078–1086, 01 2009.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, May 2017. ISSN 1095-9203. doi: 10.1126/science.aam6960. URL http://dx.doi.org/10.1126/science.aam6960.

John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences*, 36(1):48–49, 1950. doi: 10.1073/pnas.36.1.48. URL https://www.pnas.org/doi/abs/10.1073/pnas.36.1.48.

Ju Qi, Ting Feng, Falun Hei, Zhemei Fang, and Yunfeng Luo. Pure monte carlo counterfactual regret minimization.

L. J. Savage. The theory of statistical decision. *Journal of the American Statistical Association*, 46(253): 55–67, 1951. ISSN 01621459. URL http://www.jstor.org/stable/2280094.

Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007. doi: 10.1126/science.1144079. URL https://www.science.org/doi/abs/10.1126/science.1144079.

Martin Schmid, Neil Burch, Marc Lanctot, Matej Moravcík, Rudolf Kadlec, and Michael H. Bowling. Variance reduction in monte carlo counterfactual regret minimization (vr-mccfr) for extensive form games using baselines, 2018. URL https://api.semanticscholar.org/CorpusID:52180170.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

Daming Shi, Xudong Guo, Yi Liu, and Wenhui Fan. Optimal policy of multiplayer poker via actor-critic reinforcement learning. *Entropy*, 24(6), 2022. ISSN 1099-4300. doi: 10.3390/e24060774. URL https://www.mdpi.com/1099-4300/24/6/774.

Jiefu Shi and Michael L. Littman. Abstraction methods for game theoretic poker. In Tony Marsland and Ian Frank (eds.), *Computers and Games*, pp. 333–345, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45579-0.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

Prathamesh Sonawane and Arav Chheda. A survey on game theory optimal poker, 2024.

Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Pérolat, Karl Tuyls, Rémi Munos, and Michael Bowling. Actor-critic policy optimization in partially observable multiagent environments. *Advances in neural information processing systems*, 31, 2018.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA, 1998.

Oskari Tammelin. Solving large imperfect information games using cfr+, 2014.

Oskari Tammelin, Neil Burch, Michael Johanson, and Michael Bowling. Solving heads-up limit texas hold'em. In *Twenty-fourth international joint conference on artificial intelligence*, 2015.

Thomas Thompson, John Levine, and Russell Wotherspoon. Evolution of counter-strategies: Application of co-evolution to texas hold'em poker. In *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 16–22, 2008. doi: 10.1109/CIG.2008.5035616.

Christopher Watkins. Learning from delayed rewards. 01 1989.

Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992. doi: 10.1007/BF00992698.

K. Waugh, Nolan Bard, and Michael Bowling. Strategy grafting in extensive games. In *Neural Information Processing Systems*, 2009. URL https://api.semanticscholar.org/CorpusID:2946758.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. URL https://api.semanticscholar.org/CorpusID:2332513.

Jeffrey Wimpee. Finding nash equilibria in two-player, zero sum games. 2008.

Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A platform for reinforcement learning in card games. https://github.com/datamllab/rlcard, 2020a.

Daochen Zha, Kwei-Herng Lai, Songyi Huang, Yuanpu Cao, Keerthana Reddy, Juan Vargas, Alex Nguyen, Ruzhe Wei, Junyu Guo, and Xia Hu. Rlcard: A platform for reinforcement learning in card games. https://github.com/datamllab/rlcard-showdown, 2020b.

Qibin Zhou, Dongdong Bai, Junge Zhang, Fuqing Duan, and Kaiqi Huang. Decisionholdem: Safe depth-limited solving with diverse opponents for imperfect-information games, 2022.

Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In J. Platt, D. Koller, Y. Singer, and S. Roweis (eds.), *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008. URL https://proceedings.neurips.cc/paper_files/paper/2007/file/08d98638c6fcd194a4b1e6992063e944-Paper.pdf.

A Appendix

A.1 Definition of Finite Extensive Games with Imperfect Information

Definition 1 *[8, p. 200] a finite extensive game with imperfect information has the following components:*

- *A finite set N of **players**.*
- *A finite set H of sequences, the possible **histories** of actions, such that the empty sequence is in H and every prefix of a sequence in H is also in H. $Z \subseteq H$ are the terminal histories (those which are not a prefix of any other sequences). $A(h) = \{a : (h, a) \in H\}$ are the actions available after a nonterminal history $h \in H$,*
- *A function P that assigns to each nonterminal history (each member of $H \backslash Z$) a member of $N \cup \{c\}$. P is the **player function**. $P(h)$ is the player who takes an action after the history h. If $P(h) = c$ then chance determines the action taken after history h.*
- *A function f_c that associates with every history h for which $P(h) = c$ a probability measure $f_c(\cdot|h)$ on $A(h)$ ($f_c(a|h)$ is the probability that a occurs given h), where each such probability measure is independent of every other such measure.*
- *For each player $i \in N$ a partition $\mathcal{I}_i$ of $\{h \in H : P(h) = i\}$ with the property that $A(h) = A(h')$ whenever h and h' are in the same member of the partition. For $I_i \in \mathcal{I}_i$ we denote by $A(I_i)$ the set $A(h)$ and by $P(I_i)$ the player $P(h)$ for any $h \in I_i$. $\mathcal{I}_i$ is the **information partition** of player i; a set $I_i \in \mathcal{I}_i$ is an **information set** of player i.*
- *For each player $i \in N$ a utility function u_i from the terminal states Z to the reals $\mathbf{R}$. If $N = \{1, 2\}$ and $u_1 = -u_2$, it is a **zero-sum extensive game**. Define $\Delta_{u,i} = \max_z u_i(z) - \min_z u_i(z)$ to be the range of utilities to player i.*

Figure 8: A formal definition of finite extensive games with imperfect information.

A.2 Proposed Variants

Sarsa Lambda Variant is a modified version of the traditional Sarsa(λ) algorithm that maintains a separate policy dictionary mapping states to action probabilities, unlike the standard Sarsa(λ) implementation that relies solely on learned weights to estimate Q-Values. The variant updates the eligibility trace using a modified version that includes an additional term $(1 - \alpha\gamma\lambda\langle x, x\rangle)$ multiplied by the state features, whereas the traditional approach uses standard accumulating traces without this modification.

The N-Step Sarsa Variant is a hybrid method that combines elements of both N-Step Sarsa and Monte Carlo (MC) methods. It updates Q-Values based on the complete returns from each state until the end of the episode, similar to MC methods, while also incorporating the n-step lookahead aspect of N-Step Sarsa by considering the rewards and actions from the next n steps when calculating the target. Unlike traditional N-Step Sarsa, which updates Q-Values at each time step using the n-step return, the variant performs updates in batches of size n or until the end of the episode, maintaining lists of states, actions, and rewards for each batch. By combining the MC method's approach of updating Q-Values based on complete returns with the n-step lookahead of N-Step Sarsa, this hybrid method allows the agent to consider both the long-term returns and the intermediate rewards when updating Q-Values, potentially leading to more efficient and stable learning compared to pure N-Step Sarsa methods. One can find more about both of these variants in our GitHub Repository.

A.3 RLCard Leduc Hold'em Command Line Interface

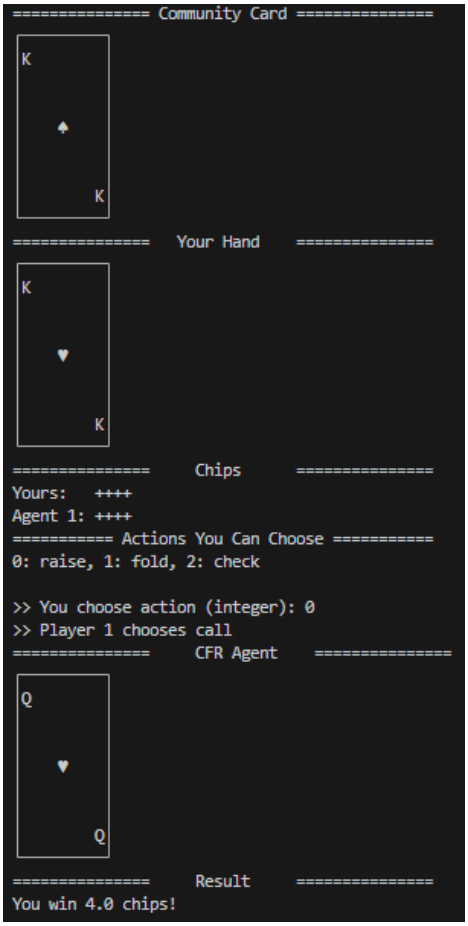

Figure 9: datamllab - RLCard Agent Playing Leduc Hold'em Intelligently (Zha et al., 2020a)