The key insight is that we’re predicting enemy/user behavior, and only after that prediction is our agent going to make a choice Also we are training our agent to most efficiently learn the enemy/user behavior using its resources, (e.g.) for music recommendations, we don’t want to explore too much and give users really different songs or they will just stop using the app, similarly in poker you don’t want to bet too much just to understand how the enemy will play Not just learn player action probability distributions as that changes for every different player, but LEARNING TO LEARN player distribution as efficiently as possible The act of using our resources efficiently between learning (explore, although it must be said that as described by the paragraph right above, learning/generalizing efficiently (learning to learn/optimizing our agent to learn well) is extremely important) and using the knowledge we learned to bet (exploiting) must be also incorporated into our continuous learning process as the end goal is to win money, learning enemy behaviors is just a step towards that (although it is a major and very important one) Maybe need a prior as well (past enemy data collection for real world at least)

For NN2, we can update the weights of our rnn/lstm every round. The reward/penalty will be based on money gained if we had purely exploited (even if agent chose to not raise because of exploration bias, if it had ignored the exploration and purely exploited, would what would have happened align with our state space) Actually should we just reveal enemy hand strength to determine how good our NN2 is, it can definitely help. See how close our prediction was to the enemy hand strength. When should we reveal enemy hand, after every turn, cycle, round? Realize that in the real game, it won’t be able to see the cards. Also, will the model update its weights at game time? Or will the weights remain unchanged after training and the model (weights) will learn to process any combination of states. The state space x1 … xn should in the end contain the strength of each players hand as an important component. If this is the case we can maybe even train after each cycle or even turn. Maybe even instead of x1 …xn being the exact estimated strength, it should be a range or maybe even a Gaussian with mean and variance, so that NN1 can better predict. Almost like a VAE This is actually allat and we can view it as a Bayesian optimization! Trying to approximate the real world with a surrogate function! For NN1 we will also update weight every round. The reward/penalty will both be based on exploitation (money gained or lost in that round and also exploration (knowledge about opponent gained Possible actions in turn: fold, call, check, raise (x money), Game vs round (pot got vs cycle (all players played once) vs turn

Also maybe a good idea to feed into NN1 a deterministic relative hand strength (so we don’t have to bother making our NN train to guess deterministic information). This relative hand strength would be calculated by (e.g. we have 0.5 and there are 2 opponents, then we have 0.5*0.5 chance of being strongest hand, 0.5*0.5*2 chance of being middle, and 0.5*0.5 chance of being worst. Binomial distribution with win or lose bring outcomes per player. 3 main ideas:

* 1. Train NN2 separately so that it learns states which should represent enemy hand strength and our agent confidence about enemy hand strength (Gaussian). Actually these states should also represent “qualities of players” such as their aggressiveness, unpredictability, patience, etc. This general characteristic prediction of enemy players along with the guess of their current hand strength should allow our agent (NN1) to predict the actions the opponents will take, and learn how to play based on the predicted actions. This is important when we want to bluff (e.g our hand strength is 0.2, we predict enemy hand strength to be 0.8, but we learned that they are not agressive, so there is a chance that he might not go all in and we can exploit that characteristic) Also we can probably use multiple neural networks, one for each enemy player
  2. Train NN1 to act not only on how to exploit what it thinks the NN1 hand is (mean of the Gaussian), but also choose actions that will reduce the variance of the Gaussian to make better predictions later.
  3. Train NN1 to also choose actions that make itself unpredictable (sample from a Gaussian of actions, with the mean of the Gaussian being the “best move” and using a variance to choose around it)

xWe are developing an intelligent agent that can robustly adapt to changing environments and adversarial agents under imperfect information in a modified version of poker (Texas hold’em). With a focus on learning and operating under imperfect information, the modified version of poker removes much of the “quantitative” aspect of poker and focuses more on the “qualitative” (bluffing, betting patterns) instead. We believe that if the environment is too focused on the strictly “optimal way to play,” for example, determining hand strength based on public knowledge, the agent will simply learn a statistical distribution rather than meaningful, imperfect information that cannot be easily statistically represented [https://poker.cs.ualberta.ca/publications/UAI05.pdf]. The paper will detail a reinforcement learning approach on learning and operating under imperfect information along with a comprehensive approach to gathering game data through game simulations and visualizations.

We have simulated a simplified version of poker by using RLCard, “a Toolkit for Reinforcement Learning in Card Games). RLCard is able to simulate a varteity of card games but critically, allows users to modify existing card games to match their required needs. Further discussion on game simulation will be discussed \_\_\_\_\_.

Each player will be assigned a random value between 0 and 1, without replacement [2], representing the strength of their hand (i.e., the percentage of other hands it beats) where 0 is the weakest and 1 is the strongest. Unlike poker, there are no public cards; the only information each player has is their own card. On each players’ turn, players will choose from four standard actions: check, call, raise, or fold. Small and big blinds (need citation), round termination, and winners will follow traditional poker regulations.

For simplicity, raises will be limited to fixed increments of 5, maxing out to (add number here). Additionally, to balance game simplicity/predictability and computational complexity/runtime, we will restrict the game to 3 players to begin with.

Within the modified version, the information of each agent can be classified as the following:

Complete Information

1. Player Card/Value

imperfect Information

1. Adversarial Card/Value
2. Adversarial Betting/Bluffing Patterns

With these areas of information, our agent will attempt to primarily exploit imperfect information to make informed decisions on the game state. Since the statistical distribution has been made relatively obvious due to the small amount of possible card combinations, the model is forced to focus on learning incomplete information rather than statistical distributions. In particular, the model will need to analyze and understand the opponent's bluffing patterns to predict when they are likely bluffing and when they are not.

Bluffing is a form misdirection in which the player intentionally provides a misrepresentation of the strength their hand. Bluffing works both ways; players can “over-represent” the strength of their hand to persuade players with stronger hands to fold or players can “under-represent/sandbag” the strength of their hand to trick players into betting higher amounts into the pot. In human-to-human play, this misrepresentation can manifest physically but qualitatively, this misrepresentation can manifest in their betting patterns.

As state by von Neumann and Morgenstern, there is a clear, important, strategic purpose to bluffing: “to create uncertainty in an opponent’s mind.” Bluffing provides a vast degree of uncertainty within the game state. If this aspect of the game is improperly captured, the model may struggle to understand certain betting patterns (ex. betting high amounts despite having a week hand). Hence, by focusing the model on learning from imperfect information, the simplified game aims to teach the model how to handle and make decisions based on incomplete or unclear data.

Poker is a extremely complicated problem for artificial intelligence due to the presence of imperfect information. With the presence of a variety of non-determinist dynamics, partial observability, and unknown adversaries, it is extremely difficult to quantify each component of the game. Current game-theory approaches, such as Nash equilibrium, are computationally expensive and even if the approximations are good, Nash equilibrium solutions are not as effective against human opponents and opponents with a clear strategy. The presence of idiosyncratic weakness in all players represents an opportunity to capitalize on a greater payoff relative to the Nash equilibrium.

To reflect the non-deterministic nature of poker, it is crucial to utilize statistical approaches to give distributions of the state of the game rather than definitive values. Whether it's deck shuffling or the imperfect information about the opponent's hand, an approach that effectively addresses the game's inherent uncertainty is crucial. Current approaches involve classifying an opponents’ hand strength in a variety of classes utilizing both Bayesian and frequentist approaches.

Our current framework involves building a virtual environment to simulate our game along with a reinforcement algorithm to facilitate agent learning. This section will outline the actions taken to develop the virtual environment and the fundamental approach to the reinforcement algorithm.

As stated in section 2, the virtual environment will be based off of RLCard, a card game simulation software developed by Zha et al (<https://arxiv.org/pdf/1910.04376>). The software toolkit has 10 pre-programmed and easily modifiable game environments with preprogrammed RL algorithms for reward agent learning. Each game environment contains 5 key files:

1. judger.py. Defines how a round terminates and how chips are distributed at the end of each round.
2. dealer.py. Defines the deck and values of cards used within the deck.
3. player.py: Defines the player state including: card in hand, amount of chips, and legal actions
4. round.py: Responsible for managing the betting round. It tracks player actions, updates the game state, and ensures that betting follows predefined rules.
5. game.py. Initializes a dealer, players, and a judger to determine the winner, with betting rounds managed by a Round object.

Instead of creating a new game environment from scratch, our group chose to modify Leduc Hold’em, a pre-existing card game that shares key characteristics with the game described in Section 2. This approach allows us to simulate our game by adjusting the five predefined files of Leduc Hold’em. Small modifications were made to each file to reflect differences in card pool (i.e using discrete values instead of a typical poker deck) and the absence of a public card (ex. actions to create public card variable or to deal a public card in game.py). With these modifications, the bluff game has been successfully simulated, as illustrated in Figure \_\_, with self-play within the RLCard framework.

REINFORCMENT LEARNING ALGORITHM  
Within the RLCard framework, there exist multiple deep-learning algorithms which can be used to facilitate agent learning. The following will outline the use of a multi-layer perceptron, Q-learning approach to reinforcement learning using the RLCard framework in a Leduc Hold’em game environment and in our modified game environment. It is important to note that the Q-learning approach is a proof-of-concept for the game to understand if our game can be learned by a reinforcement algorithm. We note that the Q-learning algorithm may not be suitable for incomplete information games; this will be explored in section 4.

The Q-learning approach is used to train agents to make optimal decisions directly engaging with the virtual environment. It involves using a Q-value to represent the expected rewards for taking an action at a specific state. The algorithm can be expressed as the following:

\[ Q(s\_t, a\_t) \leftarrow Q(s\_t, a\_t) + \alpha \left[ r\_{t+1} + \gamma \max\_{a'} Q(s\_{t+1}, a') - Q(s\_t, a\_t) \right] \] \text{where:} \begin{itemize} \item \( Q(s\_t, a\_t) \) is the Q-value for the state-action pair at time \( t \), \item \( \alpha \) is the learning rate, \item \( r\_{t+1} \) is the reward received after taking action \( a\_t \) in state \( s\_t \), \item \( \gamma \) is the discount factor, \item \( \max\_{a'} Q(s\_{t+1}, a') \) is the maximum Q-value for the next state \( s\_{t+1} \) over all possible actions \( a' \). \end{itemize}

The model implements a Deep Q-Network (DQN) agent for reinforcement learning. The agent interacts with an environment through a replay memory, storing transitions and training the network with a batch of sampled experiences. The current model employs 4 MLP blocks with outputs of 64, 128, 64, and 32 respectively, with a final linear layer to estimate the Q-value at each given state. Each MLP block contains a tanh activation function to incorporate non-linearity within the model.

With estimated Q-values, the DQN utilizes an alternative q-learning algorithm approach; epsilon-greedy. The **epsilon-greedy** algorithm is a strategy used in reinforcement learning to balance **exploration** and **exploitation**. In exploration, the agent tries random actions to discover more about the environment, while in exploitation, it selects actions that are known to maximize the expected reward based on previous experiences.

At each time step, the epsilon-greedy algorithm chooses a random action with probability epsilon (ε), and with probability 1 - ε, it chooses the action with the highest estimated value (the greedy action). The value of epsilon typically starts high to encourage exploration and decays over time to shift the focus towards exploitation as the agent learns more about the environment. This approach helps the agent avoid getting stuck in suboptimal policies while gradually refining its behavior as it gathers more knowledge. Overall, the model learns the Q-value at each state, informing the agent on which states and actions will correspond to the largest reward.

The following is the reward graph of the DQN agent in a Leduc Hold’em game against a perfectly random agent. Per RLCard, “the reward is calculated based on big blinds per hand. For example, a reward of 0.5 (-0.5) means that the player wins (loses) 0.5 times of the amount of big blind.” For our purposes, the big blind was set to 2 units. (insert image)

The DQN yielded positive results with positive reward over 1000 episodes with slight growth over multiple episodes. In a imperfect information game, the DQN agent was able to perform considerably with decent reward in a reasonable number of episodes.

4 Future Work

Our model currently adopts a Q-learning approach complemented by a fully-connected neural network. However, current literature has indicated that this approach may not be effective [sources] due to assumptions made by the agent. Consider the following example. by Brown et al. The following figure presents a modified form of Rock-Paper-Scissors (<https://arxiv.org/pdf/1805.08195>) “in which the winner receives two points (and the loser loses two points) when either player chooses scissors.” Importantly, player 2 acts after player 1 but does not observe the action of player 1, introducing imperfect information into the game.

A red circle with black text

AI-generated content may be incorrect.

Interestingly, the optimal policy for both players is to select Rock and Paper with 40% probability and Scissors with 20% probability. However, current search methods in perfect information games, such as one-ply lookahead (i.e Q-learning), fail to achieve this solution as there is insufficient information for the method to determine the optimal method. This leads to the primary challenge face by imperfect-information games: “unlike perfect-information games and single-agent settings, the value of an action may depend on the probability it is chosen.” Each state, action, and state/action sequence of operations do not have a fixed value, making it difficult to use one-ply lookahead methods. Hence, an approach which quantifies the distributive nature of the values of each state, action and sequence of state/action operations may prove useful in imperfect-information scenarios.

cd.\experiments\

mlp\_layers=[64,128,64,32], # test larger networks

For NN2, we propose updating the weights of our RNN/LSTM model after each round. The reward or penalty will be based on the money gained or lost as if the agent had purely exploited the situation, regardless of any exploration bias in its decision-making. Specifically, we could evaluate the agent’s decision by considering the hypothetical scenario in which the agent ignores exploration and purely exploits its strategy. Additionally, to assess the model's effectiveness, revealing the enemy’s hand strength could provide valuable feedback, helping to see how close our predictions were to the actual hand strength. However, a key question is when to reveal the enemy’s hand—whether it should be after every turn, cycle, or round. This is crucial because, in a real game, the agent won't have access to this information. Another consideration is whether the model should update its weights during the game or remain static after training, learning to process any combination of states based on pre-trained weights. The state space, denoted as x1...xn, should include the strength of each player's hand as a critical component. If this is the case, we could potentially update the model’s weights after each cycle or turn. Instead of representing the hand strength as an exact value, it could be modeled as a range or even a Gaussian distribution (with mean and variance), allowing NN1 to make better predictions. This approach resembles the structure of a Variational Autoencoder (VAE) and can be seen as a form of Bayesian optimization, where we try to approximate the real-world outcomes using a surrogate function. Similarly, for NN1, we will update the weights after each round, with rewards and penalties based on both exploitation (money gained or lost) and exploration (knowledge gained about the opponent). In each turn, possible actions include fold, call, check, and raise (by a certain amount). It's also important to distinguish between the different stages of the game: the turn (individual actions), round (pot contributions), and cycle (after all players have played once).

Dealer.py

Instead of using face cards, bluff game uses discrete values without suits. Dealer.py utilizes a 2-parmeter Card object, defining a suit and value for a specific card. The modified dealer.py reflects the modifications of the game by using discrete values instead of face card values and simply using the suits as additional copies of the value.

Judger.py

Since our modified game contains no public card, the judge does not need to compare each players’ hand with a public card; it only needs to check the second win condition in Leduc Hold’em, checking which player has the highest card value. The modified judger.py removes this check leaving all other code untouched.

Player.py

No changes were made as the legal actions of the player does not change between games nor does information available to the player change.

Round.py

The file is simply an initialization of the Round class provided in RLCard framework. No modifications were made to this file.

Game.py

The key difference between the two games is the absence of a public card in bluff game. Any actions to create a public card variable (line 73) or to deal a public card were removed (line 128). Other code remains the same. Players receive one card each, and small/big blinds are randomly assigned. The game supports two rounds of betting, doubling the raise amount in the second round. Actions include call, raise, fold, and check, with an optional "step back" feature to undo moves. The game ends when only one player remains or after two rounds, and payoffs are determined by a judger, normalized by the big blind.