Bluff-Bot

Alec Yu, Jason You, Arshan Ghanbari
UofT ECE 324

Abstract

Deep reinforcement learning (DRL) has achieved remarkable success in perfect-information games, yet applying these techniques to environments with hidden information and strategic deception remains an open challenge. In this work, we introduce Bluff-Bot, an intelligent agent designed to operate in Bluff Game, a simplified, variant of Texas hold'em, an incomplete information game. Unlike traditional poker AI, Bluff-Bot does not rely on predefined heuristics but instead learns to model opponent behavior and come up with near-optimal decision-making strategies under uncertainty. We first implement a baseline Deep Q-Network (DQN) agent, demonstrating that reinforcement learning can adapt to imperfect-information environments. However, due to the deterministic limitations of Q-learning, we propose a novel alternative: a Variational Autoencoder (VAE)-inspired approach with an LSTM-based continuous encoder that estimates hidden opponent states as a probabilistic latent representation. This probabilistic modeling would allow Bluff-Bot to reason about uncertainty and deploy mixed strategies, approximating Nash equilibrium solutions. This paper demonstrates the potential of probabilistic deep learning frameworks in imperfect-information settings and provides a foundation for future advancements in strategic AI decision-making.

1 Introduction

Deep reinforcement learning has demonstrated remarkable success across a wide range of domains, from mastering Atari games to conquering board games like Go. However, applying these techniques to games where agents only receive **partial information** from the environment remains a challenge. In this work, we introduce *Bluff-Bot*, an intelligent agent designed to adapt robustly to various adversarial opponents in a modified version of Texas hold'em, which we call *Bluff Game*.

With a focus on training agents to operate effectively under **incomplete information**, *Bluff Game* eliminates the need to learn intricate game-specific rules and heuristics, such as assessing the relative strengths of card combinations. This allows our agent *Bluff-Bot* to instead focus on modeling opponent behavior and making **informed decisions despite uncertainty**. By leveraging reinforcement learning and neural networks, our agent dynamically adapts to opponents, refines its strategy through experience, and optimizes decision-making in complex, hidden-information environments.

2 Problem

We have simulated a simplified version of Texas hold'em called *Bluff Game* by using *RLCard* [1], described in Section 3.1.

In *Bluff Game*, each round, every player is assigned a random integer value between 1 and 10 in the form of a card, representing the strength of their hand—where 1 is the weakest and 10 is the strongest. On their turn, which occurs multiple times within a round, players can choose from four standard actions: **check, call, raise, or fold**. The game follows traditional *Texas Hold'em* rules for small and big blinds (required initial bets), round termination, and determining the winner. Additionally, two adjustable parameters can influence the game's dynamics: the **number of rounds per game** and the **maximum number of raises per round**. Unlike *Texas Hold'em*, there are no public cards or shared values; each player only knows their own hand strength and the number of chips each player is

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

betting. To simplify the game, we assume that each player has an unlimited supply of chips. However, to prevent unrestricted betting, a fixed betting structure is imposed. Raises occur in increments of 5 chips, with a maximum bet cap of 100 per round, ensuring that the game remains focused on decision-making under uncertainty rather than resource hacking.

In this modified version, each agent's available information can be classified as follows in Table 1:

Information Type	**Details**
Complete Information	Player Card/Value
	Each Player's Chip Bet for the Round
Incomplete Information	Adversarial Card/Value
	Adversarial Betting Patterns
	Adversarial Bluffing Patterns

Table 1: Complete vs. Incomplete Information in the Game

To develop an agent that can navigate this game, we must first understand the problem better. Unlike **perfect-information games**, where all relevant details are available, *Bluff Game* introduces uncertainty through hidden information and strategic deception. Players can bluff by misrepresenting their hand strength—either by over-representing weak hands to induce folds or under-representing strong hands to extract larger bets. Since no physical tells exist in this setting, bluffing manifests purely through betting behavior [2].

This uncertainty makes poker particularly challenging for AI. Unlike deterministic games, where optimal strategies can be computed precisely, poker requires **decision-making under partial observability**. Game-theoretic approaches, such as **Nash equilibrium**, provide a theoretically optimal solution by modeling bluffing as a Bayesian inference problem. However, computing the exact Nash equilibrium is intractable in complex settings, making it impractical for real-time decision-making [3]. Instead, we aim to approximate it using a Bayesian approach, allowing the agent to infer hidden information by estimating an opponent's hand strength and playing style based on their past actions—cross-referenced with the outcomes revealed at the end of each round [4].

Given these challenges, our agent must not only estimate hidden information but also **adapt to opponent behavior**. Since betting decisions are based on uncertain information, a purely deterministic approach would fail to capture the complexity of the game. Instead, treating actions probabilistically [5] allows the model to balance exploration and exploitation, refining its decision-making over time while responding effectively to different playing styles.

3 Framework

3.1 Virtual Game Environment

To train our reinforcement learning agent, we first developed a virtual environment that simulates *Bluff Game*. This environment provides a controlled setting for agent training and evaluation, integrating a reinforcement learning framework to facilitate learning through self-play.

Our implementation is based on *RLCard*, a card game simulation toolkit developed by Zha et al. [1]. RLCard provides pre-configured environments for multiple card games and supports reinforcement learning algorithms for training agents. Each game environment in RLCard consists of five key files:

1. **judger.py**: Defines round termination conditions and reward distribution.
2. **dealer.py**: Manages deck composition, card values, and card distribution to players.
3. **player.py**: Represents player states, including hand strength, chip count, and available actions.
4. **round.py**: Oversees the betting rounds, tracking actions and updating the game state.
5. **game.py**: Initializes the game environment, including players, the dealer, and the judger, with betting rounds managed by a `Round` object.

Rather than building an environment from scratch, we modified *Leduc Hold'em*, an existing RLCard game, to align with our simplified bluffing mechanics (see Appendix A). The following summarizes the key modifications made to RLCard's five game files from Leduc Hold'em to simulate *Bluff Game*:

1. **judger.py**: Removed the public card comparison logic, as our game determines the winner solely based on private hand values.
2. **dealer.py**: Replaced face cards with integer values (1–10) and eliminated suits. The modified `Dealer.py` assigns values independently, ensuring a uniform hand strength distribution.
3. **player.py**: Removed public state from get_state method.
4. **round.py**: No modifications were made; the existing round management structure was sufficient.
5. **game.py**: Removed all mechanics related to public cards. Players receive one private card each, and small/big blinds are randomly assigned. Modified all methods related to observation space, as it has changed since there are more possible cards a player can obtain. The implemented game currently follows a two-round betting system, where the currently fixed raise amount doubles in the second round. Available actions remain standard (call, raise, fold, check), and payoffs are determined by the judger, normalized by the big blind. This will be modified later on to make the game more complex to provide a better environment for our agent.

The full implementation details, including code for the environment, dealer, player, and game logic, can be found in Appendix B.

3.2 Reinforcement Learning Algorithm

The RLCard framework provides multiple deep-learning algorithms for agent training. As a proof of concept, we implement a **Deep Q-Network (DQN)** agent using a **multi-layer perceptron (MLP)** for Q-learning. While this approach allows us to evaluate the learnability of *Bluff Game*, it may not be optimal for handling **incomplete information** games; this will be further explored in Section 5.

Q-learning trains agents by estimating **Q-values**, representing the expected reward of taking an action in a given state. The Q-value update rule is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$$

where $Q(s_t, a_t)$ is the Q-value for state-action pair (s_t, a_t), α is the learning rate, r_{t+1} is the reward, and γ is the discount factor.

Our current baseline model uses a **DQN agent** with a **replay memory** to store past transitions and train on batches of sampled experiences. The network architecture consists of **four MLP layers** with outputs of 64, 128, 64, and 32 neurons, followed by a final linear layer to estimate Q-values. Each MLP block applies a **Tanh activation function** to introduce non-linearity. To balance **exploration and exploitation**, the **epsilon-greedy strategy** [6] is used: the agent chooses a random action with probability ε and the greedy (best-estimated) action with probability $1 - \varepsilon$. The value of ε decays over time, initially favoring exploration and later shifting toward exploitation.

This approach helps prevent the agent from converging to suboptimal strategies while allowing it to refine its decision-making over time. However, Q-learning is known to struggle in **imperfect-information games** [5], as it assumes deterministic state-action evaluations, which may be insufficient for modeling hidden opponent behavior. We discuss an alternative novel approach to address this challenge in Section 5.

4 Main (Current) Results

The following figure presents the reward graph of the DQN agent in a Leduc Hold'em training run against a perfectly random agent. According to RLCard, rewards are measured in big blinds per hand, where a reward of 0.5 (-0.5) indicates that the player wins (loses) 0.5 times the big blind amount. For this experiment, we set the big blind to 2 units.

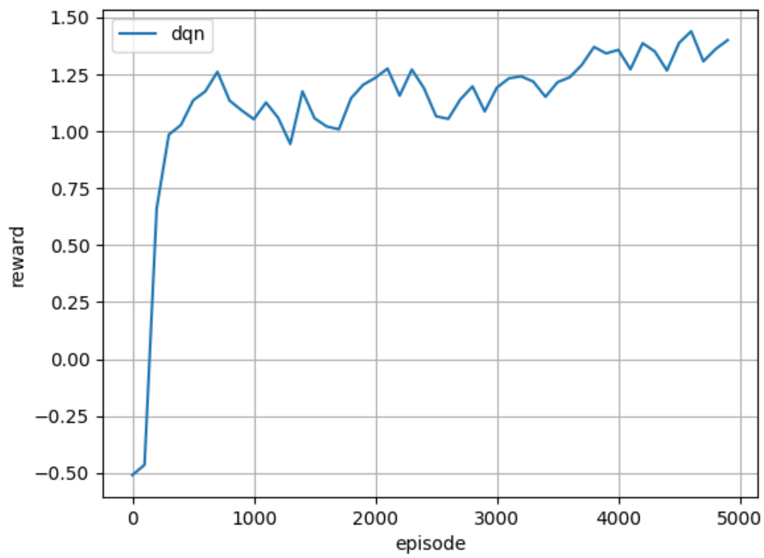


Figure 1: Reward progression for DQN agent in Leduc Hold'em training

The DQN agent demonstrated a positive learning trend over 5000 training episodes, with a gradual increase in expected rewards. Despite the inherent uncertainty in an imperfect-information game, the model achieved reasonable performance, indicating a neural network's ability to learn from opponent behavior and adjust its strategy over time.

Similarly, we conducted a training run of the DQN agent in Bluff Game, with each game consisting of a single round and allowing up to two raises per player. The resulting reward progression is illustrated in Figure 2.

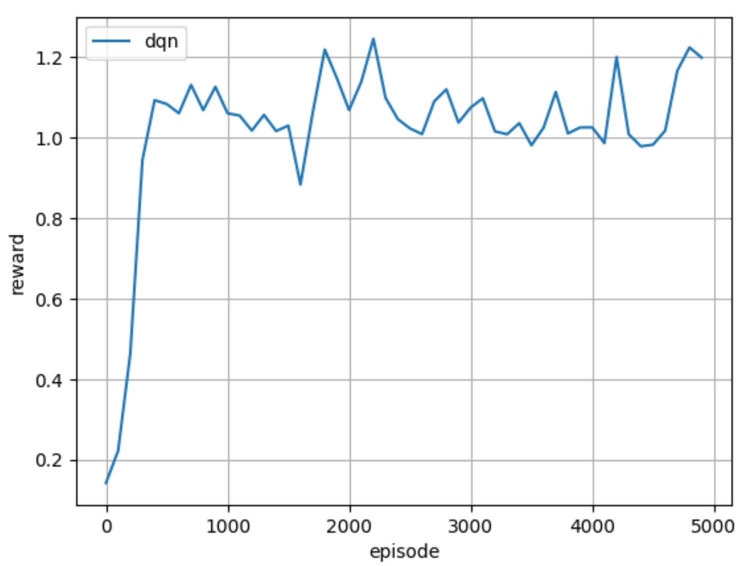


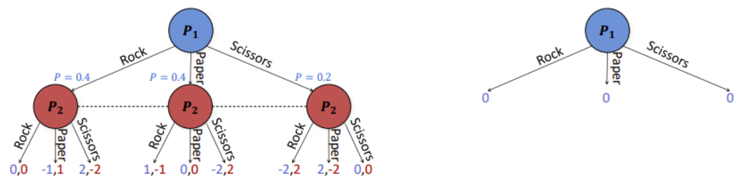
Figure 2: Reward progression for DQN agent in Bluff Game training

The performance of the DQN agent in Bluff Game exhibits a similar trend, with rewards stabilizing after an initial learning phase. While the game's lack of public information poses additional challenges, this is proof of concept that even simple neural network based agents are able to adapt over multiple training episodes and improve their decision-making under uncertainty.

For details on the implementation and training setup of these experiments, refer to Appendix C.

5 Future Work

Our model initially adopts a **Q-learning** approach with a fully connected neural network. However, prior research suggests that this method struggles in **imperfect-information** settings due to its reliance on fixed-value state-action evaluations [7]. A key example is the modified Rock-Paper-Scissors game introduced by Brown et al., where one player moves first, but their action remains hidden from the second player.



(1a) Variant of Rock-Paper-Scissors in which the optimal player 1 policy is (R=0.4, P=0.4, S=0.2). Terminal values are color-coded. The dotted lines mean player 2 does not know which node they are in.

(1b) The player 1 subgame when using perfect-information one-ply search. Leaf values are determined by the full-game equilibrium. There is insufficient information for finding (R=0.4, P=0.4, S=0.2).

Figure 3: Modified Rock-Paper-Scissors with Imperfect Information

In this scenario, the optimal strategy requires probabilistic action selection, but deterministic Q-learning fails to find this solution due to the lack of comprehensive state values. More broadly, imperfect-information games present a fundamental challenge: the value of an action is contingent on the probability that our observation is mapped to the correct perfect-information state. This makes deterministic, fixed-value learning approaches inadequate. While Q-learning provides useful heuristics for evaluating game learnability, it is insufficient for our setting. Instead, an approach that models the **distributional nature of state values** from action and observation sequences is essential for effective decision-making in imperfect-information scenarios.

To model the distribution of possible game states given our observations, we propose an approach similar to a **Variational Autoencoder (VAE)**. Our model utilizes an LSTM-based neural network as a continuous encoder, which takes as input the previously estimated state concatenated with the current observation and outputs a predicted next state. This predicted state functions analogously to the latent space in VAE architectures, where each dimension represents hidden information—with one crucial dimension encoding the opponent's hand strength and our agent's confidence in that estimate, modeled as a Gaussian distribution. Other latent dimensions can encode broader opponent characteristics, including aggressiveness, unpredictability, and patience, along with a measure of uncertainty regarding these traits. The Gaussian structure, with separate vectors for the mean and variance, allows the agent to quantify both its beliefs and the confidence in those beliefs.

When it is the agent's turn to act, it transforms the Gaussian-distributed latent state representation into a probability distribution over Q-values for each available action using an **MLP decoder**. By sampling from these Q-value distributions, the model selects an action that balances strategic exploitation with unpredictability. This formulation not only enables the agent to reason probabilistically about the uncertainty inherent in imperfect-information games but also ensures it can incorporate mixed strategies akin to Nash equilibrium solutions. Furthermore, by introducing controlled randomness into its decision-making, the agent remains difficult for adversaries to exploit, particularly those attempting to model and counter its behavior. This approach allows the agent to refine its opponent modeling efficiently while simultaneously maximizing its expected long-term rewards.

This is illustrated in Figure 4.

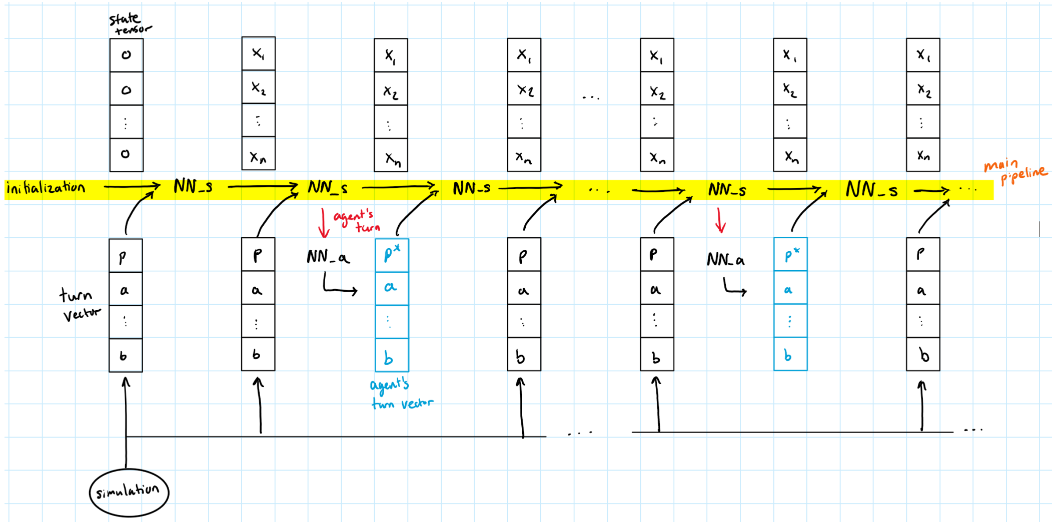


Figure 4: LSTM continuous encoder (NN_s) and MLP decoder (NN_a)

Another major benefit of this split architecture is its ability to train more efficiently by incorporating **intermediate reward signals** throughout the game. Unlike traditional single-network approaches such as DQN, which typically rely on delayed rewards at the end of a round or game, our framework allows the encoder-decoder structure to be rewarded dynamically. Specifically, the encoding network can be incentivized for accurately predicting opponent hand states while accounting for uncertainty — where reduced uncertainty is rewarded more heavily. Additionally, we can guide the model to balance exploration and exploitation during its training more effectively; if an action leads to a meaningful refinement of the latent state representation, it signals that the model has gained useful information, justifying an additional reward even though taking such an action may result in a small short-term punishment. This approach enables the agent to learn and converge more efficiently toward an approximation of the Nash equilibrium strategy. While the exact Nash equilibrium for complex games like this may not be fully determined, our method provides a structured way to approximate it as closely as possible, improving both robustness and adaptability to adversarial opponents.

References

[1] Daochen Zha, Zhiwei Lin, Junyu Guo, Zhengyu Zhou, Zihan Tang, Jinjun Song, Xia Hu, and Chandra Reddy. Rlcard: A toolkit for reinforcement learning in card games. *Proceedings of the 2019 Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[2] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1953.

[3] George Musser. The nash equilibrium is the optimal poker strategy—expert players don't always use it. *Scientific American*, 2010.

[4] Darse Billings, Neil Burch, Aaron Davidson, Robert C Holte, Jonathan Schaeffer, Duane Szafron, and Michael Bowling. Bayes' bluff: Opponent modeling in poker. *Proceedings of the 21st Conference on Uncertainty in Artificial Intelligence (UAI)*, 2005.

[5] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *arXiv preprint arXiv:2007.13544*, 2020.

[6] Michel Tokic and Günther Palm. Epsilon-greedy q-learning algorithms for reinforcement learning. *arXiv preprint arXiv:1911.09891*, 2019.

[7] Noam Brown and Tuomas Sandholm. Depth-limited solving for imperfect-information games. *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Scope Change Statement.

We are not changing the scope of our proposal. The context and goals of our project remain as originally outlined in the initial proposal.

A Leduc Hold'em Overview

Leduc Hold'em is a simplified version of *Limit Texas Hold'em*, introduced in *Bayes' Bluff: Opponent Modeling in Poker* [4]. It uses a **six-card deck**, consisting of two copies of each rank (King, Queen, Jack), and features two betting rounds. Each game follows these steps:

1. Each player posts an initial blind bet and is dealt one private card.
2. Players place bets in the first round.
3. A single public card is revealed.
4. Players place bets in the second round.
5. The player whose private card matches the public card wins; otherwise, the higher-ranked private card wins.

We used Leduc Hold'em as the basis for our game environment due to its **structured betting mechanics** and **reduced game complexity**, making it an ideal starting point for modifying into *Bluff Game*.

B Bluff Game Implementation Details

This section provides the implementation details of *Bluff Game*, including the game environment, dealer, player, and round management. The following code snippets outline our modifications to the RLCard framework:

B.1 Game Environment

```
# game.py - Contains classes necessary to initialize game
    environment

class BluffGame(Game):

    def __init__(self, allow_step_back=False, num_players=2):
        ''' Initialize the class bluffgame Game
        '''
        self.allow_step_back = allow_step_back
        self.np_random = np.random.RandomState()
        ''' No big/small blind
        # Some configarations of the game
        # These arguments are fixed in Bluff-Game

        # Raise amount and allowed times
        self.raise_amount = 2
        self.allowed_raise_num = 2

        self.num_players = 2
        '''
        # Some configarations of the game
        # These arguments can be specified for creating new games

        # Small blind and big blind
        self.small_blind = 1
        self.big_blind = 2 * self.small_blind

        # Raise amount and allowed times
        self.raise_amount = self.big_blind
        self.allowed_raise_num = 2

        self.num_players = num_players

    def configure(self, game_config):
```

```
        ''' Specifiy some game specific parameters, such as number
            of players
        '''
        self.num_players = game_config['game_num_players']

    def init_game(self):
        ''' Initialilze the game of Limit Texas Hold'em

        This version supports two-player limit texas hold'em

        Returns:
            (tuple): Tuple containing:

                (dict): The first state of the game
                (int): Current player's id
        '''
        # Initilize a dealer that can deal cards
        self.dealer = Dealer(self.np_random)

        # Initilize two players to play the game
        self.players = [Player(i, self.np_random) for i in range(
            self.num_players)]

        # Initialize a judger class which will decide who wins in
            the end
        self.judger = Judger(self.np_random)

        # Prepare for the first round
        for i in range(self.num_players):
            self.players[i].hand = self.dealer.deal_card()
        # Randomly choose a small blind and a big blind
        s = self.np_random.randint(0, self.num_players)
        b = (s + 1) % self.num_players
        self.players[b].in_chips = self.big_blind
        self.players[s].in_chips = self.small_blind

        # The player with small blind plays the first
        self.game_pointer = s

        # Initilize a bidding round, in the first round, the big
            blind and the small blind needs to
        # be passed to the round for processing.
        self.round = Round(raise_amount=self.raise_amount,
                           allowed_raise_num=self.
                               allowed_raise_num,
                           num_players=self.num_players,
                           np_random=self.np_random)

        self.round.start_new_round(game_pointer=self.game_pointer,
             raised=[p.in_chips for p in self.players])

        # Count the round. There are 2 rounds in each game.
        self.round_counter = 0

        # Save the hisory for stepping back to the last state.
        self.history = []

        state = self.get_state(self.game_pointer)
```

```
        return state, self.game_pointer

    def step(self, action):
        ''' Get the next state

        Args:
            action (str): a specific action. (call, raise, fold,
                or check)

        Returns:
            (tuple): Tuple containing:

                (dict): next player's state
                (int): next plater's id
        '''
        if self.allow_step_back:
            # First snapshot the current state
            r = copy(self.round)
            r_raised = copy(self.round.raised)
            gp = self.game_pointer
            r_c = self.round_counter
            d_deck = copy(self.dealer.deck)
            ps = [copy(self.players[i]) for i in range(self.
                num_players)]
            ps_hand = [copy(self.players[i].hand) for i in range(
                self.num_players)]
            self.history.append((r, r_raised, gp, r_c, d_deck, ps,
                 ps_hand))

        # Then we proceed to the next round
        self.game_pointer = self.round.proceed_round(self.players,
             action)

        # If a round is over, ...
        if self.round.is_over():
            #Double the raise amount for the second round
            if self.round_counter == 0:
                self.round.raise_amount = 2 * self.raise_amount

            self.round_counter += 1
            self.round.start_new_round(self.game_pointer)

        state = self.get_state(self.game_pointer)

        return state, self.game_pointer

    def get_state(self, player):
        ''' Return player's state

        Args:
            player_id (int): player id

        Returns:
            (dict): The state of the player
        '''
        chips = [self.players[i].in_chips for i in range(self.
            num_players)]
        legal_actions = self.get_legal_actions()
```

```
        state = self.players[player].get_state(chips,
            legal_actions)
        state['current_player'] = self.game_pointer

        return state

    def is_over(self):
        ''' Check if the game is over

        Returns:
            (boolean): True if the game is over
        '''
        alive_players = [1 if p.status=='alive' else 0 for p in
            self.players]
        # If only one player is alive, the game is over.
        if sum(alive_players) == 1:
            return True

        # If all rounds are finshed
        if self.round_counter >= 2:
            return True
        return False

    def get_payoffs(self):
        ''' Return the payoffs of the game

        Returns:
            (list): Each entry corresponds to the payoff of one
                player
        '''
        chips_payoffs = self.judger.judge_game(self.players)
        payoffs = np.array(chips_payoffs) / (self.big_blind)
        return payoffs

    def step_back(self):
        ''' Return to the previous state of the game

        Returns:
            (bool): True if the game steps back successfully
        '''
        if len(self.history) > 0:
            self.round, r_raised, self.game_pointer, self.
                round_counter, d_deck, self.players, ps_hand =
                self.history.pop()
            self.round.raised = r_raised
            self.dealer.deck = d_deck
            for i, hand in enumerate(ps_hand):
                self.players[i].hand = hand
            return True
        return False

# bluffgame.py - Initializes the game environment

DEFAULT_GAME_CONFIG = {
        'game_num_players': 2,
        }

class BluffGameEnv(Env):
    ''' bluff-game Environment
```

```
    '''

    def __init__(self, config):
        ''' Initialize the Limitholdem environment
        '''
        self.name = 'bluffgame'
        self.default_game_config = DEFAULT_GAME_CONFIG
        self.game = Game()
        super().__init__(config)
        self.actions = ['call', 'raise', 'fold', 'check']
        self.state_shape = [[40] for _ in range(self.num_players)]
             #? what is 40 --> observation size
        self.action_shape = [None for _ in range(self.num_players)
            ]

        with open(os.path.join(rlcard.__path__[0], 'games/
            bluffgame/card2index.json'), 'r') as file:
             self.card2index = json.load(file)

    def _get_legal_actions(self):
        ''' Get all leagal actions

        Returns:
            encoded_action_list (list): return encoded legal
                action list (from str to int)
        '''
        return self.game.get_legal_actions()

    def _extract_state(self, state):
        ''' Extract the state representation from state dictionary
             for agent

        Args:
            state (dict): Original state from the game

        Returns:
            observation (list): combine the player's score and
                dealer's observable score for observation
        '''
        extracted_state = {}

        legal_actions = OrderedDict({self.actions.index(a): None
            for a in state['legal_actions']})
        extracted_state['legal_actions'] = legal_actions

        hand = state['hand']
        obs = np.zeros(40) # Changed all obs
        obs[self.card2index[hand]] = 1
        obs[state['my_chips']+10] = 1
        obs[sum(state['all_chips'])-state['my_chips']+25] = 1 #???
        extracted_state['obs'] = obs

        extracted_state['raw_obs'] = state
        extracted_state['raw_legal_actions'] = [a for a in state['
            legal_actions']]
        extracted_state['action_record'] = self.action_recorder

        return extracted_state
```

```
    def get_payoffs(self):
        ''' Get the payoff of a game

        Returns:
           payoffs (list): list of payoffs
        '''
        return self.game.get_payoffs()

    def _decode_action(self, action_id):
        ''' Decode the action for applying to the game

        Args:
            action id (int): action id

        Returns:
            action (str): action for the game
        '''
        legal_actions = self.game.get_legal_actions()
        if self.actions[action_id] not in legal_actions:
            if 'check' in legal_actions:
                return 'check'
            else:
                return 'fold'
        return self.actions[action_id]

    def get_perfect_information(self):
        ''' Get the perfect information of the current state

        Returns:
            (dict): A dictionary of all the perfect information of
                 the current state
        '''
        state = {}
        state['chips'] = [self.game.players[i].in_chips for i in
           range(self.num_players)]
        state['hand_cards'] = [self.game.players[i].hand.get_index
           () for i in range(self.num_players)]
        state['current_round'] = self.game.round_counter
        state['current_player'] = self.game.game_pointer
        state['legal_actions'] = self.game.get_legal_actions()
        return state
```

B.2 Dealer Implementation

```
# dealer.py - Handles card distribution
class BluffDealer(Dealer):

    def __init__(self, np_random):
        ''' Initialize a bluff-game dealer class
        '''
        self.np_random = np_random
        self.deck = [Card('S', 'A'), Card('S', '2'), Card('S',
            '3'), Card('S', '4'), Card('S', '5'), Card('S', '6'),
           Card('S', '7'), Card('S', '8'), Card('S', '9'), Card('
           S', 'T'),
                     Card('H', 'A'), Card('H', '2'), Card('H',
                        '3'), Card('H', '4'), Card('H', '5'),
                        Card('H', '6'), Card('H', '7'), Card('H',
                          '8'), Card('H', '9'), Card('H', 'T')]
        self.shuffle()
```

```
        self.pot = 0
```

B.3 Player Implementation

```
# player.py - Defines player state
class BluffPlayer:

    def __init__(self, player_id, np_random):
        ''' Initilize a player.

        Args:
            player_id (int): The id of the player
        '''
        self.np_random = np_random
        self.player_id = player_id
        self.status = 'alive'
        self.hand = None

        # The chips that this player has put in until now
        self.in_chips = 0

    def get_state(self, all_chips, legal_actions):
        ''' Encode the state for the player

        Args:
            all_chips (int): The chips that all players have put
                in

        Returns:
            (dict): The state of the player
        '''
        # removed public state from get_state method
        state = {}
        state['hand'] = self.hand.get_index()
        state['all_chips'] = all_chips
        state['my_chips'] = self.in_chips
        state['legal_actions'] = legal_actions
        return state

    def get_player_id(self):
        ''' Return the id of the player
        '''
        return self.player_id
```

B.4 Round Management

```
# round.py - Manages betting rounds
class BluffRound(Round):
    ''' Round can call other Classes' functions to keep the game
       running
    '''

    def __init__(self, raise_amount, allowed_raise_num,
       num_players, np_random):
        ''' Initilize the round class

        Args:
            raise_amount (int): the raise amount for each raise
            allowed_raise_num (int): The number of allowed raise
               num
```

```
        num_players (int): The number of players
    '''
    super(BluffRound, self).__init__(raise_amount,
        allowed_raise_num, num_players, np_random=np_random)
```

These modifications create a simplified environment that retains the fundamental aspects of bluffing, allowing us to focus on solving the problem of decision making under uncertainty instead of being sidetracked by game-specific mechanics.

C DQN Training Implementation

This section provides the code used for training the DQN agent in both Leduc Hold'em and Bluff Game. The implementation follows the RLCard framework with modifications for our custom environment.

C.1 Training Script

```
''' Training DQN agent on custom bluff-game environment
'''

import os
import argparse

import torch

import rlcard
from rlcard.agents import RandomAgent
from rlcard.utils import (
    get_device,
    set_seed,
    tournament,
    reorganize,
    Logger,
    plot_curve,
)
def train(args):

    # Check whether gpu is available
    device = get_device()

    # Seed numpy, torch, random
    set_seed(args.seed)

    # Make the environment with seed
    env = rlcard.make(
        args.env,
        config={
            'seed': args.seed,
        }
    )

    # Initialize the agent and use random agents as opponents
    if args.algorithm == 'dqn':
        from rlcard.agents import DQNAgent
        agent = DQNAgent(
            num_actions=env.num_actions,
            state_shape=env.state_shape[0],
            mlp_layers=[64,128,64,32], # test larger networks
            device=device,
        )
```

```
    elif args.algorithm == 'nfsp':
        from rlcard.agents import NFSPAgent
        agent = NFSPAgent(
            num_actions=env.num_actions,
            state_shape=env.state_shape[0],
            hidden_layers_sizes=[64,64],
            q_mlp_layers=[64,64],
            device=device,
        )
    agents = [agent]
    for _ in range(1, env.num_players):
        agents.append(RandomAgent(num_actions=env.num_actions))
    env.set_agents(agents)

    # Start training
    with Logger(args.log_dir) as logger:
        for episode in range(args.num_episodes):

            if args.algorithm == 'nfsp':
                agents[0].sample_episode_policy()

            # Generate data from the environment
            trajectories, payoffs = env.run(is_training=True)

            # Reorganaize the data to be state, action, reward,
                next_state, done
            trajectories = reorganize(trajectories, payoffs)

            # Feed transitions into agent memory, and train the
                agent
            # Here, we assume that DQN always plays the first
                position
            # and the other players play randomly (if any)
            for ts in trajectories[0]:
                agent.feed(ts)

            # Evaluate the performance. Play with random agents.
            if episode % args.evaluate_every == 0:
                logger.log_performance(
                    episode,
                    tournament(
                        env,
                        args.num_eval_games,
                    )[0]
                )

        # Get the paths
        csv_path, fig_path = logger.csv_path, logger.fig_path

    # Plot the learning curve
    plot_curve(csv_path, fig_path, args.algorithm)

    # Save model
    save_path = os.path.join(args.log_dir, 'model.pth')
    torch.save(agent, save_path)
    print('Model saved in', save_path)

if __name__ == '__main__':
    parser = argparse.ArgumentParser("DQN/NFSP example in RLCard")
```

```
parser.add_argument(
    '--env',
    type=str,
    default='bluffgame',
    choices=[
        'blackjack',
        'leduc-holdem',
        'limit-holdem',
        'doudizhu',
        'mahjong',
        'no-limit-holdem',
        'uno',
        'gin-rummy',
        'bridge',
    ],
)
parser.add_argument(
    '--algorithm',
    type=str,
    default='dqn',
    choices=[
        'dqn',
        'nfsp',
    ],
)
parser.add_argument(
    '--cuda',
    type=str,
    default='',
)
parser.add_argument(
    '--seed',
    type=int,
    default=42,
)
parser.add_argument(
    '--num_episodes',
    type=int,
    default=5000, # change for more epochs
)
parser.add_argument(
    '--num_eval_games',
    type=int,
    default=2000, # change for better eval
)
parser.add_argument(
    '--evaluate_every',
    type=int,
    default=100,
)
parser.add_argument(
    '--log_dir',
    type=str,
    default='experiments/bluffgame_dqn_result/',
)

args = parser.parse_args()

os.environ["CUDA_VISIBLE_DEVICES"] = args.cuda
```

```
    train(args)
```

C.2 Agent Architecture

```
class DQNAgent(object):
    '''
    Approximate clone of rlcard.agents.dqn_agent.DQNAgent
    that depends on PyTorch instead of Tensorflow
    '''
    def __init__(self,
                 replay_memory_size=20000,
                 replay_memory_init_size=100,
                 update_target_estimator_every=1000,
                 discount_factor=0.99,
                 epsilon_start=1.0,
                 epsilon_end=0.1,
                 epsilon_decay_steps=20000,
                 batch_size=32,
                 num_actions=2,
                 state_shape=None,
                 train_every=1,
                 mlp_layers=None,
                 learning_rate=0.00005,
                 device=None,
                 save_path=None,
                 save_every=float('inf'),):

        '''
        Q-Learning algorithm for off-policy TD control using
            Function Approximation.
        Finds the optimal greedy policy while following an epsilon
            -greedy policy.

        Args:
            replay_memory_size (int): Size of the replay memory
            replay_memory_init_size (int): Number of random
                experiences to sample when initializing
              the reply memory.
            update_target_estimator_every (int): Copy parameters
                from the Q estimator to the
              target estimator every N steps
            discount_factor (float): Gamma discount factor
            epsilon_start (float): Chance to sample a random
                action when taking an action.
              Epsilon is decayed over time and this is the start
                  value
            epsilon_end (float): The final minimum value of
                epsilon after decaying is done
            epsilon_decay_steps (int): Number of steps to decay
                epsilon over
            batch_size (int): Size of batches to sample from the
                replay memory
            evaluate_every (int): Evaluate every N steps
            num_actions (int): The number of the actions
            state_space (list): The space of the state vector
            train_every (int): Train the network every X steps.
            mlp_layers (list): The layer number and the dimension
                of each layer in MLP
            learning_rate (float): The learning rate of the DQN
                agent.
```

```
            device (torch.device): whether to use the cpu or gpu
            save_path (str): The path to save the model
                checkpoints
            save_every (int): Save the model every X training
                steps
        '''
        self.use_raw = False
        self.replay_memory_init_size = replay_memory_init_size
        self.update_target_estimator_every =
            update_target_estimator_every
        self.discount_factor = discount_factor
        self.epsilon_decay_steps = epsilon_decay_steps
        self.batch_size = batch_size
        self.num_actions = num_actions
        self.train_every = train_every

        # Torch device
        if device is None:
            self.device = torch.device('cuda:0' if torch.cuda.
                is_available() else 'cpu')
        else:
            self.device = device

        # Total timesteps
        self.total_t = 0

        # Total training step
        self.train_t = 0

        # The epsilon decay scheduler
        self.epsilons = np.linspace(epsilon_start, epsilon_end,
            epsilon_decay_steps)

        # Create estimators
        self.q_estimator = Estimator(num_actions=num_actions,
            learning_rate=learning_rate, state_shape=state_shape,
            \
            mlp_layers=mlp_layers, device=self.device)
        self.target_estimator = Estimator(num_actions=num_actions,
             learning_rate=learning_rate, state_shape=state_shape,
             \
            mlp_layers=mlp_layers, device=self.device)

        # Create replay memory
        self.memory = Memory(replay_memory_size, batch_size)

        # Checkpoint saving parameters
        self.save_path = save_path
        self.save_every = save_every

    def feed(self, ts):
        ''' Store data in to replay buffer and train the agent.
            There are two stages.
            In stage 1, populate the memory without training
            In stage 2, train the agent every several timesteps

        Args:
            ts (list): a list of 5 elements that represent the
                transition
```

```
        '''
        (state, action, reward, next_state, done) = tuple(ts)
        self.feed_memory(state['obs'], action, reward, next_state
            ['obs'], list(next_state['legal_actions'].keys()),
            done)
        self.total_t += 1
        tmp = self.total_t - self.replay_memory_init_size
        if tmp>=0 and tmp%self.train_every == 0:
            self.train()

    def step(self, state):
        ''' Predict the action for genrating training data but
            have the predictions disconnected from the computation
                graph

        Args:
            state (numpy.array): current state

        Returns:
            action (int): an action id
        '''
        q_values = self.predict(state)
        epsilon = self.epsilons[min(self.total_t, self.
            epsilon_decay_steps-1)]
        legal_actions = list(state['legal_actions'].keys())
        probs = np.ones(len(legal_actions), dtype=float) * epsilon
             / len(legal_actions)
        best_action_idx = legal_actions.index(np.argmax(q_values))
        probs[best_action_idx] += (1.0 - epsilon)
        action_idx = np.random.choice(np.arange(len(probs)), p=
            probs)

        return legal_actions[action_idx]

    def eval_step(self, state):
        ''' Predict the action for evaluation purpose.

        Args:
            state (numpy.array): current state

        Returns:
            action (int): an action id
            info (dict): A dictionary containing information
        '''
        q_values = self.predict(state)
        best_action = np.argmax(q_values)

        info = {}
        info['values'] = {state['raw_legal_actions'][i]: float(
            q_values[list(state['legal_actions'].keys())[i]]) for
            i in range(len(state['legal_actions']))}

        return best_action, info

    def predict(self, state):
        ''' Predict the masked Q-values

        Args:
            state (numpy.array): current state
```

```
        Returns:
            q_values (numpy.array): a 1-d array where each entry
                represents a Q value
        '''

        q_values = self.q_estimator.predict_nograd(np.expand_dims(
            state['obs'], 0))[0]
        masked_q_values = -np.inf * np.ones(self.num_actions,
            dtype=float)
        legal_actions = list(state['legal_actions'].keys())
        masked_q_values[legal_actions] = q_values[legal_actions]

        return masked_q_values

    def train(self):
        ''' Train the network

        Returns:
            loss (float): The loss of the current batch.
        '''
        state_batch, action_batch, reward_batch, next_state_batch,
             done_batch, legal_actions_batch = self.memory.sample
            ()

        # Calculate best next actions using Q-network (Double DQN)
        q_values_next = self.q_estimator.predict_nograd(
            next_state_batch)
        legal_actions = []
        for b in range(self.batch_size):
            legal_actions.extend([i + b * self.num_actions for i
                in legal_actions_batch[b]])
        masked_q_values = -np.inf * np.ones(self.num_actions *
            self.batch_size, dtype=float)
        masked_q_values[legal_actions] = q_values_next.flatten()[
            legal_actions]
        masked_q_values = masked_q_values.reshape((self.batch_size
            , self.num_actions))
        best_actions = np.argmax(masked_q_values, axis=1)

        # Evaluate best next actions using Target-network (Double
           DQN)
        q_values_next_target = self.target_estimator.
            predict_nograd(next_state_batch)
        target_batch = reward_batch + np.invert(done_batch).astype
            (np.float32) * \
             self.discount_factor * q_values_next_target[np.arange(
                self.batch_size), best_actions]

        # Perform gradient descent update
        state_batch = np.array(state_batch)

        loss = self.q_estimator.update(state_batch, action_batch,
            target_batch)
        print('\rINFO - Step {}, rl-loss: {}'.format(self.total_t,
             loss), end='')

        # Update the target estimator
        if self.train_t % self.update_target_estimator_every == 0:
```

```
            self.target_estimator = deepcopy(self.q_estimator)
            print("\nINFO - Copied model parameters to target
                network.")

        self.train_t += 1

        if self.save_path and self.train_t % self.save_every == 0:
            # To preserve every checkpoint separately,
            # add another argument to the function call
                parameterized by self.train_t
            self.save_checkpoint(self.save_path)
            print("\nINFO - Saved model checkpoint.")


    def feed_memory(self, state, action, reward, next_state,
        legal_actions, done):
        ''' Feed transition to memory

        Args:
            state (numpy.array): the current state
            action (int): the performed action ID
            reward (float): the reward received
            next_state (numpy.array): the next state after
                performing the action
            legal_actions (list): the legal actions of the next
                state
            done (boolean): whether the episode is finished
        '''
        self.memory.save(state, action, reward, next_state,
            legal_actions, done)

    def set_device(self, device):
        self.device = device
        self.q_estimator.device = device
        self.target_estimator.device = device

    def checkpoint_attributes(self):
        '''
        Return the current checkpoint attributes (dict)
        Checkpoint attributes are used to save and restore the
            model in the middle of training
        Saves the model state dict, optimizer state dict, and all
            other instance variables
        '''

        return {
            'agent_type': 'DQNAgent',
            'q_estimator': self.q_estimator.checkpoint_attributes
                (),
            'memory': self.memory.checkpoint_attributes(),
            'total_t': self.total_t,
            'train_t': self.train_t,
            'epsilon_start': self.epsilons.min(),
            'epsilon_end': self.epsilons.max(),
            'epsilon_decay_steps': self.epsilon_decay_steps,
            'discount_factor': self.discount_factor,
            'update_target_estimator_every': self.
                update_target_estimator_every,
            'batch_size': self.batch_size,
```

```
            'num_actions': self.num_actions,
            'train_every': self.train_every,
            'device': self.device
        }

    @classmethod
    def from_checkpoint(cls, checkpoint):
        '''
        Restore the model from a checkpoint

        Args:
            checkpoint (dict): the checkpoint attributes generated
                by checkpoint_attributes()
        '''

        print("\nINFO - Restoring model from checkpoint...")
        agent_instance = cls(
            replay_memory_size=checkpoint['memory']['memory_size
                '],
            update_target_estimator_every=checkpoint['
               update_target_estimator_every'],
            discount_factor=checkpoint['discount_factor'],
            epsilon_start=checkpoint['epsilon_start'],
            epsilon_end=checkpoint['epsilon_end'],
            epsilon_decay_steps=checkpoint['epsilon_decay_steps'],
            batch_size=checkpoint['batch_size'],
            num_actions=checkpoint['num_actions'],
            device=checkpoint['device'],
            state_shape=checkpoint['q_estimator']['state_shape'],
            mlp_layers=checkpoint['q_estimator']['mlp_layers'],
            train_every=checkpoint['train_every']
        )

        agent_instance.total_t = checkpoint['total_t']
        agent_instance.train_t = checkpoint['train_t']

        agent_instance.q_estimator = Estimator.from_checkpoint(
            checkpoint['q_estimator'])
        agent_instance.target_estimator = deepcopy(agent_instance.
            q_estimator)
        agent_instance.memory = Memory.from_checkpoint(checkpoint
            ['memory'])


        return agent_instance

    def save_checkpoint(self, path, filename='checkpoint_dqn.pt'):
        ''' Save the model checkpoint (all attributes)

        Args:
            path (str): the path to save the model
        '''
        torch.save(self.checkpoint_attributes(), path + '/' +
            filename)

class Estimator(object):
    '''
    Approximate clone of rlcard.agents.dqn_agent.Estimator that
```

```
uses PyTorch instead of Tensorflow.  All methods input/output
   np.ndarray.

Q-Value Estimator neural network.
This network is used for both the Q-Network and the Target
   Network.
'''

def __init__(self, num_actions=2, learning_rate=0.001,
   state_shape=None, mlp_layers=None, device=None):
    ''' Initilalize an Estimator object.

    Args:
        num_actions (int): the number output actions
        state_shape (list): the shape of the state space
        mlp_layers (list): size of outputs of mlp layers
        device (torch.device): whether to use cpu or gpu
    '''
    self.num_actions = num_actions
    self.learning_rate=learning_rate
    self.state_shape = state_shape
    self.mlp_layers = mlp_layers
    self.device = device

    # set up Q model and place it in eval mode
    qnet = EstimatorNetwork(num_actions, state_shape,
       mlp_layers)
    qnet = qnet.to(self.device)
    self.qnet = qnet
    self.qnet.eval()

    # initialize the weights using Xavier init
    for p in self.qnet.parameters():
        if len(p.data.shape) > 1:
            nn.init.xavier_uniform_(p.data)

    # set up loss function
    self.mse_loss = nn.MSELoss(reduction='mean')

    # set up optimizer
    self.optimizer =  torch.optim.Adam(self.qnet.parameters(),
        lr=self.learning_rate)

def predict_nograd(self, s):
    ''' Predicts action values, but prediction is not included
        in the computation graph.  It is used to predict
           optimal next
        actions in the Double-DQN algorithm.

    Args:
      s (np.ndarray): (batch, state_len)

    Returns:
      np.ndarray of shape (batch_size, NUM_VALID_ACTIONS)
         containing the estimated
      action values.
    '''
    with torch.no_grad():
        s = torch.from_numpy(s).float().to(self.device)
```

```
		q_as = self.qnet(s).cpu().numpy()
	return q_as

def update(self, s, a, y):
	''' Updates the estimator towards the given targets.
		In this case y is the target-network estimated
		value of the Q-network optimal actions, which
		is labeled y in Algorithm 1 of Minh et al. (2015)

	Args:
	  s (np.ndarray): (batch, state_shape) state
	     representation
	  a (np.ndarray): (batch,) integer sampled actions
	  y (np.ndarray): (batch,) value of optimal actions
	     according to Q-target

	Returns:
	  The calculated loss on the batch.
	'''
	self.optimizer.zero_grad()

	self.qnet.train()

	s = torch.from_numpy(s).float().to(self.device)
	a = torch.from_numpy(a).long().to(self.device)
	y = torch.from_numpy(y).float().to(self.device)

	# (batch, state_shape) -> (batch, num_actions)
	q_as = self.qnet(s)

	# (batch, num_actions) -> (batch, )
	Q = torch.gather(q_as, dim=-1, index=a.unsqueeze(-1)).
	   squeeze(-1)

	# update model
	batch_loss = self.mse_loss(Q, y)
	batch_loss.backward()
	self.optimizer.step()
	batch_loss = batch_loss.item()

	self.qnet.eval()

	return batch_loss

def checkpoint_attributes(self):
	''' Return the attributes needed to restore the model from
		a checkpoint
	'''
	return {
		'qnet': self.qnet.state_dict(),
		'optimizer': self.optimizer.state_dict(),
		'num_actions': self.num_actions,
		'learning_rate': self.learning_rate,
		'state_shape': self.state_shape,
		'mlp_layers': self.mlp_layers,
		'device': self.device
	}

@classmethod
```

```
    def from_checkpoint(cls, checkpoint):
        ''' Restore the model from a checkpoint
        '''
        estimator = cls(
            num_actions=checkpoint['num_actions'],
            learning_rate=checkpoint['learning_rate'],
            state_shape=checkpoint['state_shape'],
            mlp_layers=checkpoint['mlp_layers'],
            device=checkpoint['device']
        )

        estimator.qnet.load_state_dict(checkpoint['qnet'])
        estimator.optimizer.load_state_dict(checkpoint['optimizer
            '])
        return estimator


class EstimatorNetwork(nn.Module):
    ''' The function approximation network for Estimator
        It is just a series of tanh layers. All in/out are torch.
            tensor
    '''

    def __init__(self, num_actions=2, state_shape=None, mlp_layers
        =None):
        ''' Initialize the Q network

        Args:
            num_actions (int): number of legal actions
            state_shape (list): shape of state tensor
            mlp_layers (list): output size of each fc layer
        '''
        super(EstimatorNetwork, self).__init__()

        self.num_actions = num_actions
        self.state_shape = state_shape
        self.mlp_layers = mlp_layers

        # build the Q network
        layer_dims = [np.prod(self.state_shape)] + self.mlp_layers
        fc = [nn.Flatten()]
        fc.append(nn.BatchNorm1d(layer_dims[0]))
        for i in range(len(layer_dims)-1):
            fc.append(nn.Linear(layer_dims[i], layer_dims[i+1],
                bias=True))
            fc.append(nn.Tanh())
        fc.append(nn.Linear(layer_dims[-1], self.num_actions, bias
            =True))
        self.fc_layers = nn.Sequential(*fc)

    def forward(self, s):
        ''' Predict action values

        Args:
            s  (Tensor): (batch, state_shape)
        '''
        return self.fc_layers(s)

class Memory(object):
```

```
    ''' Memory for saving transitions
    '''

    def __init__(self, memory_size, batch_size):
        ''' Initialize
        Args:
            memory_size (int): the size of the memroy buffer
        '''
        self.memory_size = memory_size
        self.batch_size = batch_size
        self.memory = []

    def save(self, state, action, reward, next_state,
       legal_actions, done):
        ''' Save transition into memory

        Args:
            state (numpy.array): the current state
            action (int): the performed action ID
            reward (float): the reward received
            next_state (numpy.array): the next state after
                performing the action
            legal_actions (list): the legal actions of the next
                state
            done (boolean): whether the episode is finished
        '''
        if len(self.memory) == self.memory_size:
            self.memory.pop(0)
        transition = Transition(state, action, reward, next_state,
            done, legal_actions)
        self.memory.append(transition)

    def sample(self):
        ''' Sample a minibatch from the replay memory

        Returns:
            state_batch (list): a batch of states
            action_batch (list): a batch of actions
            reward_batch (list): a batch of rewards
            next_state_batch (list): a batch of states
            done_batch (list): a batch of dones
        '''
        samples = random.sample(self.memory, self.batch_size)
        samples = tuple(zip(*samples))
        return tuple(map(np.array, samples[:-1])) + (samples[-1],)

    def checkpoint_attributes(self):
        ''' Returns the attributes that need to be checkpointed
        '''

        return {
            'memory_size': self.memory_size,
            'batch_size': self.batch_size,
            'memory': self.memory
        }

    @classmethod
    def from_checkpoint(cls, checkpoint):
        '''
```

```
        Restores the attributes from the checkpoint

        Args:
            checkpoint (dict): the checkpoint dictionary

        Returns:
            instance (Memory): the restored instance
        '''

        instance = cls(checkpoint['memory_size'], checkpoint['
            batch_size'])
        instance.memory = checkpoint['memory']
        return instance
```

This script trains a DQN agent for 5000 episodes and evaluates performance against a random agent. The model architecture consists of four MLP layers with Tanh activations, followed by a final linear layer predicting Q-values for each action.

NeurIPS Paper Checklist

The checklist is designed to encourage best practices for responsible machine learning research, addressing issues of reproducibility, transparency, research ethics, and societal impact. Do not remove the checklist: **The papers not including the checklist will be desk rejected.** The checklist should follow the references and follow the (optional) supplemental material. The checklist does NOT count towards the page limit.

Please read the checklist guidelines carefully for information on how to answer these questions. For each question in the checklist:

- You should answer [Yes] , [No] , or [NA] .
- [NA] means either that the question is Not Applicable for that particular paper or the relevant information is Not Available.
- Please provide a short (1–2 sentence) justification right after your answer (even for NA).

The checklist answers are an integral part of your paper submission. They are visible to the reviewers, area chairs, senior area chairs, and ethics reviewers. You will be asked to also include it (after eventual revisions) with the final version of your paper, and its final version will be published with the paper.

The reviewers of your paper will be asked to use the checklist as one of the factors in their evaluation. While "[Yes] " is generally preferable to "[No] ", it is perfectly acceptable to answer "[No] " provided a proper justification is given (e.g., "error bars are not reported because it would be too computationally expensive" or "we were unable to find the license for the dataset we used"). In general, answering "[No] " or "[NA] " is not grounds for rejection. While the questions are phrased in a binary way, we acknowledge that the true answer is often more nuanced, so please just use your best judgment and write a justification to elaborate. All supporting evidence can appear either in the main paper or the supplemental material, provided in appendix. If you answer [Yes] to a question, in the justification please point to the section(s) where related material for the question can be found.

IMPORTANT, please:

- **Delete this instruction block, but keep the section heading "NeurIPS paper checklist"**,
- **Keep the checklist subsection headings, questions/answers and guidelines below.**
- **Do not modify the questions and only use the provided macros for your answers**.

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: We assert that our primary claim—developing a deep RL agent (Bluff-Bot) capable of learning and adapting to imperfect-information gameplay in Bluff Game—is clearly stated in the introduction and substantiated through our framework description and preliminary experimental results. We justify our contributions by comparing against standard baselines and detailing our modifications to the RLCard toolkit.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We acknowledge that while our report demonstrates proof-of-concept success with a DQN agent, there are limitations. These include the inherent challenges of deterministic Q-learning in imperfect-information settings and a need for more extensive statistical validation (e.g., error bars, confidence intervals). We plan to address these in future revisions.

3. **Theory Assumptions and Proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: Although our work is primarily empirical and does not provide formal proofs, we discuss the theoretical limitations of deterministic Q-learning in approximating Nash equilibrium strategies in imperfect-information games. Our report motivates the need for a probabilistic approach—specifically, an encoder–decoder framework—that better models the distribution of possible states and actions, aligning more closely with the mixed strategies inherent in Nash equilibrium.

4. **Experimental Result Reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results?

   Answer: [Yes]

   Justification: We describe the environment setup, hyperparameters (e.g., the four-layer MLP with neuron counts of 64, 128, 64, and 32, Tanh activations, epsilon–greedy exploration, etc.), and training protocols in Sections 3.1, 3.2, and Appendices B and C. However, our current results are preliminary, and we plan to include additional run statistics and error measurements in the final submission.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We have included the DQN agent architecture code along with detailed modifications to the RLCard toolkit in the report appendices, notably in Appendix B, which contains the code for the modified game environment and agent training scripts. Moving forward, we plan to upload the complete modified environment files to further enhance reproducibility. Additionally, repository links and comprehensive documentation will be provided as supplementary material.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Our report includes detailed descriptions of the virtual game environment, agent architecture, and training setup. While most details are provided, we will add further specifics (such as optimizer settings and complete hyperparameter ranges) in our final version to ensure all experimental protocols are fully reproducible.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: Our preliminary reward graphs demonstrate positive trends; however, we have not yet reported error bars or confidence intervals. In our final submission, we will conduct multiple runs and report appropriate statistical measures to confirm the significance of our results.

8. **Experiments Compute Resources**

Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

Answer: [No]

Justification: Although we describe the experimental setup and number of episodes, we have not explicitly detailed the compute resources (e.g., GPU/CPU models, memory, training time). We will include these details in the final version.

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Our work adheres to standard ethical guidelines for academic research. As our study is focused on simulation and algorithmic development without any direct human subject impact, it aligns with accepted research ethics practices.

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [No]

    Justification: While our paper focuses extensively on the technical development and empirical evaluation of Bluff-Bot in a simulated environment, it does not explicitly discuss the broader societal impacts of our work. We have not yet explored potential risks or benefits (e.g., ethical implications, real-world applicability, or unintended consequences) associated with deploying game-playing AI. In light of this, we plan to evaluate both the potential benefits (such as advancing research in opponent modeling and reinforcement learning) and the risks (such as misuse in contexts like real-world gambling) to fully address the broader impacts of our work.

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [Yes]

    Justification: Given that our research is confined to simulated environments and does not involve sensitive real-world data, the risks of misuse are low. Nonetheless, we note that any potential adaptation for real-world applications would require additional safeguards.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We have built our work on the publicly available RLCard toolkit and other open-source resources. We have cited these assets appropriately and will ensure that all licensing terms are respected when releasing our code and data.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [NA]

    Justification: Our paper does not introduce new datasets or proprietary assets beyond the modifications we have made to existing tools.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: Our research involves algorithm development and simulation only, with no crowdsourced data or human subjects involved.