Cribbage AI agent

Ashwin Sharan and Pravin Anand Pawar

[sharan.a, pawar.prav]@northeastern.edu

Abstract

Non-deterministic imperfect information games pose challenges for Artificial Intelligence (AI) design, as compared to AI for perfect information games.[3]For the purpose of this paper we use two techniques one Monte Carlo Tree Search (MCTS), an AI technique that uses random sampling of game playouts to build a search tree rather than domain specific knowledge about how to play a given game, has been used successfully in some perfect information games and the other Double Deep Q-Network(Double DQN), a model free algorithm which uses neutral network as function approximators for Q-values, has performed well in several games. We will also apply DRQN algorithms and analyse if learning from history helps in this cases. MCTS has beed implemented using techniques including sampling over many determinizations of a starting game state, and considering which information set the player belongs to.

Introduction

Reinforcement learning(RL) is gaining a lot of attention in recent times. Using RL algorithms, researchers are able to beat the human level of intelligence in many games. For example, AlphaZero has performed well in the GO game and is also able to beat the human level of intelligence. Similarly, using neural networks as functional approximators has also achieved good performance in many atari games.

Card games are one of the famous topics in research. Lots of research are done for playing card games like poker, blackjack, etc. Open source environment toolkit are also created for simulation card games for Reinforcement learning research. For example, RL card toolkit for game environments, which aided increase in research in card games.

In this project, we are focusing on analyzing and exploring performance of AI agents, implementing MCTS, DDQN and DRQN algorithm, to play card game name cribbage, against traditional random and greedy agents. Since cribbage is imperfect information game, as the opponents cannot see each others cards, it will difficult to determine goal state as the opponent moves can impact how to get to goal states resulting in increase complexity in finding optimal strategy to play card.

Background

Cribbage

Cribbage is a card game for 2-4 players using a standard 52 card deck. For this project, only the 2-player version of the game was considered. Cribbage is played in repeated hands until a player wins by reaching 121 points at any time during a game. In this paper we only work with the part playing aspect of the game. In each hand, players are each dealt six cards, and both discard two cards face down to a form a third hand (the crib), which is scored by the dealer at the end of the hand. In the play stage of the hand, players alternate playing cards from their hand face up while keeping a running sum of the ranks of cards played. When no one can play without going over 31, the play restarts from zero with the remaining cards. When all cards are played, players count the points earned by their hands, and the dealer also counts the crib as their second hand. Players score points in both the play and counting stages by getting sums of 15 and 31, pairs, flushes, straights (runs), and some other combinations. Cribbage, with only 13 cards in play per hand, has a small game tree for a given deal of the cards. If Cribbage were to be played with all cards face-up as a perfect information game, each hand would have at most 129600 possible combinations.[3]

MCTS

MCTS was first described in. In MCTS a search tree is built by repeatedly playing out game simulations with random moves and recording the average win rate of different moves. MCTS builds the game tree asymmetrically by focusing on more promising branches. Every node of the search tree in MCTS accumulates information about how successful it has been in previous iterations. That information is then used to bias the selection of child nodes at every level of the search in subsequent search iterations. MCTS is an on-line search–nothing needs to be precomputed to use MCTS–and it is an anytime search, meaning it can be stopped whenever a computational or time budget is reached. When the search is stopped, the best move found so far from the root of the tree is selected. In more detail, the procedure for building the tree is as follows:

- **Nodes:** The tree consists of nodes representing states in the game. Each node keeps track of its visit count and

total score or value (win or loss in most games) from visiting that node, as well as a reference to its parent node and child nodes.

- **Selection:** Descend the tree from the root node by following a selection policy until either a terminal node or a node with unexpanded children is reached.
- **Expansion:** If the selected node is not a terminal node, expand it by creating a new node representing an action taken from the parent node and the state arrived at by taking that action.
- **Simulation:**Play from the expanded node by following a default policy until reaching a terminal game state, which has a value (score for Cribbage) for each player associated with it. The default policy is usually random play but can be otherwise.
- **Backpropagation:**Backup the simulation values to all the nodes visited in the selection and expansion steps.[3]

This a typical structure of MCTS but for this paper we make use of a modified approach to deal with a partially observable environment.

Single observer Information set MCTS

Terminologies used in the section related to the concempts:

- Derterminization: It is a process of making a partially observable environment fully observable by creating a Information set of all possible state including the opponents moves and creating a tree for each state depicting a play out. It is an expensive process with both time and memory. For a environment like card game it would look like all player can see all cards at all states of the game.
- Information set: It is a set of all possible states for a partially observable environment it is used by determinization for creating trees.

To overcome the problems associated with the determinization approach, we propose searching a single tree whose nodes correspond to information sets rather than states. In single-observer information set MCTS (SO-ISMCTS), nodes in the tree correspond to information sets from the root player's point of view, and edges correspond to actions (i.e., moves from the point of view of the player who plays them). The correspondence between nodes and information sets is not one–one: partially observable opponent moves that are indistinguishable to the root player have separate edges in the tree, and thus the resulting information set has several nodes in the tree.

Fig. 1 shows a game tree for a simple single-player game of imperfect information. The root information set contains two states: x and y. The player first selects one of two actions: a_1 or a_2 . Selecting a_2 yields an immediate reward of $+0.5$ and ends the game. If the player instead selects a_1, he must then select an action a_3 or a_4. If the game began in state , then and lead to rewards -1 of and $+1$, respectively (this information being revealed by means of environment action $e_{x!3}$ or $e_{x!4}$); if the game began in state y, then the rewards are interchanged.[4]

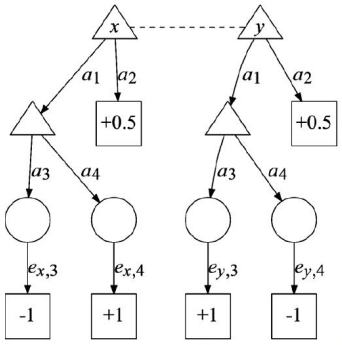


Fig. 1. A game tree for a simple single-player game. Nodes represent game states. Nodes shaped △ denote player 1 decision states, ◯ environment states, and □ terminal states labeled with reward values for player 1. Nonterminal nodes in corresponding positions in the x and y subtrees are in the same player 1 information set; this is shown by a dashed line for the root nodes. Adapted from [18, Fig. 1].

If states x and y are equally likely, action a_1 has an expectimax value of 0: upon choosing a_1, both a_2 and a_4 have an expectimax value of 0. Thus, the optimal action fromthe root is a_2. However, a determinizing player searches trees corresponding to each state x and y individually and assigns a_1 a minimax value of $+1$ in each (by assuming that the correct choice of a_3 or a_4 can always be made), thus believing a_1 to be optimal. This is an example of strategy fusion.

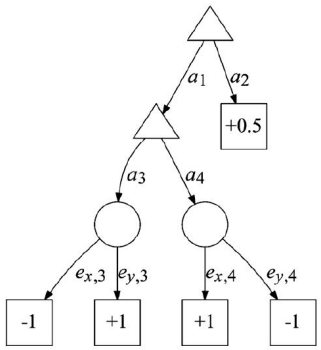


Fig. 2. An information set search tree for the game shown in Fig. 1. Here nodes shaped △ denote information sets where player 1 is both the observer and the player about to act.

Fig. 2 shows the tree searched by SO-ISMCTS for this game. In this case, each node is in one–one correspondence with an information set. After a sufficiently large number of iterations the algorithm assigns each environment node an expected value of 0 and thus assigns the same value to action a_1, thus overcoming strategy fusion and correctly identifying a_2 as the optimal move.[4]

The above traversal is an example of how the SOISMCTS applys to a situation. In the case of our cribbage environment a determinization is created before each iteration of the search. As the selection, expansion, and simulation steps are conducted for an iteration, only actions which are compatible with the current determinization are considered. Node selection is based on how good an action has been in previous determinizations that included the same action as a possibility.[4]

In Cribbage, the actions available to the player to act in a given node are the same in all determinizations in which that node is reachable, because that players cards remain the same in all determinizations. The opponent may have certain actions available from a given node in one determinization that are not available in another determinization. In, the selection formula for SO-IS MCTS is modified by considering only the number of times a node has been available for selection, rather than the number of times its parent node has been selected. In normal UCT those numbers are the same, but in SO-IS MCTS, actions that are rarely available would be over-selected if the number of visits to the parent were used in the second term of the formula.[4] The modified formula is:

$$argmax_{c \in c(v,d)} \left(\frac{r(c)_{\rho(d)}}{n(c)} + k\sqrt{\frac{\log n'(c)}{n(c)}} \right)$$

- $r(c)$ = reward value of child node.
- $n(c)$ = visit count of the child node.
- $\rho(d)$ = viable action for the determinization.
- $n'(c)$ = availability count of the child.
- d = determinization.
- $c(v,d) = \{u \in c(v) : a(u) \in A(d)\}$,the children of v compatible with determinization d.
- k = hyper parameter to adjust exploration expoitation.

Here is a simple view algorithm of the SO-ISMCTS for a more descriptive one look in the Appendix.

Algorithm 1: High-level pseudocode for the SO-ISMCTS algorithm. More detailed pseudocode is given in part A of the Appendix. For the variant of this algorithm with partially observable moves (SO-ISMCTS+POM) simply replace the word "action" below with "move (from player 1's viewpoint) and see the more detailed pseudocode in part B of the Appendix.

```
1:  function SO-ISMCTS([s_0]^~1 , n)
2:      create a single-node tree with root v_0 corresponding to the
        root information set [s_0]^~1 (from player 1's viewpoint)
3:      for n iterations do
4:          choose a determinization d at random from [s_0]^~1, and
            use only nodes/actions compatible with d this iteration
5:
6:          //Selection
7:          repeat
8:              descend the tree (restricted to nodes/actions
                compatible with d) using the chosen bandit algorithm
9:          until a node v is reached such that some action from
            v leads to a player 1 information set which is not
            currently in the tree or until v is terminal
10:
11:         //Expansion
12:         if v is nonterminal then
13:             choose at random an action a from node v that is
                compatible with d and does not exist in the tree
14:             add a child node to v corresponding to the player
                1 information set reached using action a and set
                it as the new current node v
15:
16:         //Simulation
17:         run a simulation from v to the end of the game using
            determinization d
18:
19:         //Backpropagation
20:         for each node u visited during this iteration do
21:             update u's visit count and total simulation reward
22:             for each sibling w of u that was available for
                selection when u was selected, including u itself do
23:                 update w's availability count
24:
25:     return an action from the root node v_0 such that the
        number of visits to the corresponding child node is
        maximal
```

Double DQN and DRQN

Q-Learning is a model free reinforcement learning algorithm used to find best action in a given state. It internally uses Q-table to store the value of actions in different states. But in real world applications state space and action space can be huge and it might not be possible to store all state spaces and compute efficiently. Since tabular q-learning is not scalable, DQN will be helpful for large state space. Tabular Q-Learning update equation:

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

where s is current state, a is current action, s' is resulting state and a' is action from resulting state.

DQN is also a model-free algorithm that combines Q-Learning with deep neural networks, used as a functional approximation for Q-table. It outputs parameterized Q-values of actions for given state s. Deep neural network help DQN receive the benefits of fast computation using less memory as well as generalization capability, learning from given experiences and applying that learning to similar situations. Thus, if some unseen state appears in a game, a naive algorithm might fail. But, DQN using the deep neural network will still be able to provide better results based on its generalized learning from past experiences.

Algorithm 1 : Double Q-learning (Hasselt et al., 2015)

Initialize primary network Q_θ, target network $Q_{\theta'}$, replay buffer $\mathcal{D}$, $\tau << 1$
for each iteration **do**
  for each environment step **do**
    Observe state s_t and select $a_t \sim \pi(a_t, s_t)$
    Execute a_t and observe next state s_{t+1} and reward $r_t = R(s_t, a_t)$
    Store (s_t, a_t, r_t, s_{t+1}) in replay buffer $\mathcal{D}$
  for each update step **do**
    sample $e_t = (s_t, a_t, r_t, s_{t+1}) \sim \mathcal{D}$
    Compute target Q value:
      $Q^*(s_t, a_t) \approx r_t + \gamma\, Q_\theta(s_{t+1}, argmax_{a'} Q_{\theta'}(s_{t+1}, a'))$
    Perform gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
    Update target network parameters:

[1]

Q-Learning is also know to face maximization bias because of overestimation of q-value. Thus, Double DQN will be helpful to mitigate maximization bias issue. DDQN has also performed well in large number of atari games. Double DQN uses two different network behavior net and target network.

Remembering the history might provide some benefit in imperfect information space and Recurrent neural networks are best choice. Thus, DRQN algorithm, which is replacing first layer of DQN by LSTM layer, might provide some benefits in learning.

Related Work

Here are listed alternatives technique that can be taken to approach this problem:

Cheating UCT The simplest version of a MCTS Cribbage player agent we implemented is a cheating player. The cheating player plays the game as a perfect information game. It has access to all cards, including unrevealed cards in the desk. The Cheating UCT agent ignores the issue of imperfect information, but it is a useful benchmark for other AI players.
Cheat UCT did not seem like a cogent approach to solve our problem statement of assessing performance in a partially observable environment for an RL agent.

Determinized UCT On any players move when there is hidden information in the game state, from that players perspective they may be in any one of many possible game states. That combination of states together form an information set for that player. A determinization of a game state is any state from the players information set. Perfect Information Monte Carlo (PIMC) is used to play games. PIMC involves repeatedly taking determinizations from the information set the player to act is in, and then using a standard AI technique for playing out that determinization as a perfect information game. The move chosen at the end of the search is either the one with the highest average value or the one with the most visits over all explored determinizations.

The visit counts of all child nodes of the root node are summed across all determinizations, and the action corresponding to the child node with the most visits is returned as best move by the Determinized UCT agent. To create a determinization, each part of the game state which the current player has seen is held fixed, and the rest of the game state is randomized.
Determinized UCT is too cumbersome as it would create a tree for all possible states consuming a vast section of memory and time to give a result.

Neural Fictitious Self-Play (NFSP)

NFSP is design to deal with problems of large-scale games of imperfect information. This algorithms combine fictitious self-play with deep neural network. This algorithms was able to learn strategy which shown state-of-the-art performance in poker. It is computationally heavy and requires lot of training iteration to show good performance.

Project description

For this project assignment we worked on creating a card game agent for cribbage. Cribbage is a two-player, competitive, round-based card game split into two distinct phases, a discard and peg phase. Here is a quick overview of the game (link given at the end of the document).
For a test environment we will be making use of an open source text based cribbage interface found from a git repository (link given at the end of the document) To experiment and generate episodes our RL agents we will be making use of naive greedy agent and random agent.
A project like this has been done before but using different approaches and with very little success.

Random Agent

Our most simple agent solves the discard problem by randomly discarding two cards to the crib. During pegging, the random agent randomly picks a card to play from its hand. While this behavior is not particularly "intelligent", it serves as a baseline level of play to evaluate our other agents against. Ideally, all of our agents should outscore the random agent.

Greedy Agent

The Greedy agent will operate by simply making the best choice to push its own score up at every instant, regardless of scoring opportunities that such behavior might open up for the opponent. For instance, for the discard problem the greedy agent simply computes the value of all the possible resulting hands (plus the score of whatever it puts in the crib

if the agent is the dealer) and chooses the option that maximizes this value. For the pegging problem, it chooses two cards at random.

SO-ISMCTS Agent

In the game environment obtained from the git repository we implement a new class of player in player.py called *singleObserverISMCTS* based on the on the algorithm given below.

function SO-ISMCTS($[s_0]^{\sim_1}, n$)
  create a single-node tree with root v_0 corresponding to $[s_0]^{\sim_1}$
  for n iterations **do**
    choose $d_0 \in [s_0]^{\sim_1}$ uniformly at random
    $(v, d) \leftarrow$ SELECT(v_0, d_0)
    if $u(v, d) \neq \emptyset$ **then**
      $(v, d) \leftarrow$ EXPAND(v, d)
    $r \leftarrow$ SIMULATE(d)
    BACKPROPAGATE(r, v)
  return $a(c)$ where $c \in \arg\max_{c \in c(v_0)} n(c)$

function SELECT(v, d)
  while d is nonterminal and $u(v, d) = \emptyset$ **do**
    select$^3 c \in \arg\max_{c \in c(v,d)} \left(\frac{r(c)_{\rho(d)}}{n(c)} + k\sqrt{\frac{\log n'(c)}{n(c)}} \right)$
    $v \leftarrow c$; $d \leftarrow f(d, a(c))$
  return (v, d)

function EXPAND(v, d)
  choose a from $u(v, d)$ uniformly at random
  add a child w to v with $a(w) = a$
  $v \leftarrow w$;$d \leftarrow f(d, a)$
  return (v, d)

function SIMULATE(d)
  while d is nonterminal **do**
    choose a from $A(d)$ uniformly at random
    $d \leftarrow f(d, a)$
  return $\mu(d)$

function BACKPROPAGATE(r, v_l)
  for each node v from v_l to v_0 **do**
    increment $n(v)$ by 1
    $r(v) \leftarrow r(v) + r$
    let d_v be the determinization when v was visited
    for each sibling w of v compatible with d_v, including v itself **do**
      increment $n'(w)$ by 1

- $r(c)$ = reward value of child node.
- $n(c)$ = visit count of the child node.
- $\rho(d)$ = viable action for the determinization.
- $n'(c)$ = availability count of the child.
- d = determinization.
- $c(v)$ = children of node v.
- $a(v)$ = incoming action at node v.
- $c(v, d) = \{u \in c(v) : a(u) \in A(d)\}$,the children of v compatible with determinization d.
- k = hyper parameter to adjust exploration and exploitation for this case we are taking it as $\sqrt{2}$.[4]
- $u(v, d) = \{a \in A(d) : c \in c(v, d) with a(c) = a\}$ the actions from d for which v does not have children in the current tree. Note that $c(v, d)$ and $u(v, d)$ are defined only for and such that is a determinization of (i.e., a state contained in) the information set to which corresponds.

In, the selection formula for SO-ISMCTS is modified by considering only the number of times a node has been available for selection, rather than the number of times its parent node has been selected. In normal UCT those numbers are the same, but in SO-ISMCTS, actions that are rarely available would be over-selected if the number of visits to the parent were used in the second term of the formula.[3] The modified formula is:

$$argmax_{c \in c(v,d)} \left(\frac{r(c)_{\rho(d)}}{n(c)} + k\sqrt{\frac{\log n'(c)}{n(c)}} \right)$$

MDP for the SOISMCTS agent Here is a description of how the markove decision process works for as follows:

- **State** An entier game state is preserved in a determinization as shown in the code sip-pit below.

```
class determinization:
    def __init__(self, state=None, play=0, count=0):
        self.state = state # [[Cards][Table cards][Enemy Cards]]
        self.play = play #[Cards no longer being counted]
        self.count = count #Count of cards
        self.TableP1 = []  #Cards already played
        self.TableP2 = []  #Cards played by the enemy
        self.go = False # Go has been said or not.
```

- **Transition function** Its the same one as the game environment it just modifies the ”determinzation” variable.
- **Action** The actions are the card value, the game is set in such a way that players are not allowed to perform any other action like ’go’.
- **Reward** Here the reward is the difference in the player score and enemy score at the end of the hand.

Information set To run this algoritm we need a Information set of all possible moves at each step of the for which we create the following function.

```
Here we create a n-possible game state based on what we
already know about the game.
"""
def InformationSet(self, hand):
    self.enemyTable = []
    deck = Deck()
    state = []
    InfoSet = []
    # We record the seen cards that are excluded from creating a
    # combination.
    seenCards = hand + self.plays + self.crib
    seenCards.append(self.turn_card)
    # We calulate the number of cards in enemy hand.
    enemyCardCount = 8 - len(hand) - len(self.plays)
    # We create a list of unseen cards.
    unseenCards = [n for n in deck.cards if n not in seenCards]
    # If enemy has no cards to play there is only one possible state.
    if enemyCardCount <= 0:
        state.append(hand)
        state.append(self.plays)
        state.append([])
        InfoSet.append(state)
    else:
        # We create a compbination of unseen cards the enemy can
        # have.
        for subset in combinations(unseenCards, enemyCardCount):
            state.append(hand)
            state.append(self.plays)
            state.append(list(subset))
            InfoSet.append(state)
            state = []
```

Double DQN and DRQN Agent

In this part, we have created 2 player one implementing DDQN and other implementing DRQN algorithm to selection an action and play against random and greedy agent as opponent. Double DQN and DRQN implementation of players are similar to standard implementations.

DQN:

```
class DQN(nn.Module):
    def __init__(self, input_size, hidden_layer_size, output_size):
        super(DQN, self).__init__()
        self.inputLayer = nn.Linear(input_size, hidden_layer_size)
        self.relu_fn_1 = nn.ReLU()
        self.hiddenLayer1 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.relu_fn_2 = nn.ReLU()
        self.hiddenLayer2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.relu_fn_3 = nn.ReLU()
        self.outputLayer = nn.Linear(hidden_layer_size, output_size)

    def forward(self, x):
        x = self.inputLayer(x)
        x = self.relu_fn_1(x)
        x = self.hiddenLayer1(x)
        x = self.relu_fn_2(x)
        x = self.hiddenLayer2(x)
        x = self.relu_fn_3(x)
        y = self.outputLayer(x)
        return y
```

Double DQN Agent: Internally using DQN for behavior and target network.

```
# Double DQN Agent
class DDQNAgent(object):
    def __init__(self, parameters):
        self.params = parameters
        self.action_dim = parameters['action_dimension']
        self.obs_dim = parameters['observation_dimension']
        self.behavior_network = DQN(input_size=parameters['observation_dimension'],
                                    hidden_layer_size=parameters['hidden_layer_dimension'],
                                    output_size=parameters['action_dimension'])
        self.target_network = DQN(input_size=parameters['observation_dimension'],
                                  hidden_layer_size=parameters['hidden_layer_dimension'],
                                  output_size=parameters['action_dimension'])

        self.optimizer = torch.optim.Adam(self.behavior_network.parameters(), lr=parameters['learning_rate'])
        self.device = "cpu"
        self.gamma = 0.99
```

DRQN:

```
class DRQN(nn.Module):
    def __init__(self, input_size, hidden_layer_size, output_size):
        super(DRQN, self).__init__()
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_layer_size,
                            num_layers=1)
        self.relu_fn_1 = nn.ReLU()
        self.hiddenLayer1 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.relu_fn_2 = nn.ReLU()
        self.hiddenLayer2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.relu_fn_3 = nn.ReLU()
        self.outputLayer = nn.Linear(hidden_layer_size, output_size)

    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.relu_fn_1(x)
        x = self.hiddenLayer1(x)
        x = self.relu_fn_2(x)
        x = self.hiddenLayer2(x)
        x = self.relu_fn_3(x)
        y = self.outputLayer(x)
        return y
```

DRQN Agent:

```
class DRQNAgent(object):
    def __init__(self, parameters):
        self.params = parameters
        self.gamma = 1
        self.action_dim = parameters['action_dimension']
        self.obs_dim = parameters['observation_dimension']
        self.behavior_network = DRQN(input_size=parameters['observation_dimension'],
                                     hidden_layer_size=parameters['hidden_layer_dimension'],
                                     output_size=parameters['action_dimension'])
        self.target_network = DRQN(input_size=parameters['observation_dimension'],
                                   hidden_layer_size=parameters['hidden_layer_dimension'],
                                   output_size=parameters['action_dimension'])
        self.behavior_network.eval()
        self.optimizer = torch.optim.Adam(self.behavior_network.parameters(), lr=parameters['learning_rate'])
        self.device = "cpu"
```

Update Equation for behavior net:

$$w \leftarrow w - \alpha \nabla_w L(B; w, w^-)$$

Update Equation for Target net:

$$w^- \leftarrow w$$

Target :

$$target(s', a'; w, w^-) = r + \gamma Q_{w^-}(s', \arg\max_{a'} Q_w(s', a'))$$

Loss :

$$\nabla_w L(B; w, w^-) \approx -\frac{1}{|B|} \sum_{(s,a,s',r) \in B} \left(target(s', a'; w, w^-) - Q_w(s, a)\right) \nabla_w Q_w(s, a)$$

Agent Transition: Observation after card '1' is played by AI agent and '8' by opponent is shown below. Integer values in below transition represents card number which is been played and 0 represents no card placed on table or card not present in hand.

- **Observation(features/state):** (cards on table, cards in hand)

  For example:

cards on table (8) : 5, 9, 0, 0, 0, 0, 0, 0
cards in hand(4): 1, 2, 7, 0
Observation: (5, 9, 0, 0, 0, 0, 0, 0, 1, 2, 7, 0)

- **Action:** (select card from cards in hand)

  For example,
  cards in hand: (1, 2, 7, 0)
  Selected action: 1
  "Go" action value is 14
  (Action is "Go" when there is no card to play, card count on table cannot exceed 31)

- **Next observation(next state):**
  (cards on table, cards in hand)

  Next observation: (5, 9, 1, 8, 0, 0, 0, 0, 2, 7, 0, 0)
- **Reward:**
  Difference between score of player and its opponent.

States(observations) are converted into one hot vector and passed to neural network for ease of learning. Parameter of neural network:

```
parameters = {
            "action_dimension": 15 * 1,
            "observation_dimension": 13 * 14,
            "hidden_layer_dimension": 256,
            "learning_rate": 0.1,
            "epsilon": 0.1,
            "gamma": 0.99,
            "behaviorNetworkFreq": 4,
            "targetUpdateFreq": 1000
        }
```

Experiments

MCTS

For the first part of this experiment we test how it performs against the random agent and greedy agent for creating a tree with 1000 node possible moves for each game state, where we play a total of 400 games to assess its performance. It makes sense for the scope of this experiment because This experiment is cogent as it should show signs of learning and should give a fair win-percentage.

Player1	Player2	Win Percentage of player 2
Random Agent	SOISMCTS	54%
Greedy Agent	SOISMCTS	48.66%
Random Agent	Greedy Agent	51%

The above experiment was completed 2hrs 13 mins due the the large number of games and and each step of the game required a tree to be created with a 1000 nodes. According to the table we have good cribbage agent as it at least or around 50% of the time, as some of the best players in history will have win percentages over 55% because of the random nature of the game [5]. Now two avenues were explored:

– To change with the UCB selection or node selection to see if a higher win percentage can be obtained.
– To see if we can get a better win rate by increasing the number of nodes possible.

For this part of the experiment we only make use of the random agent to create a baseline.

Changing the UCB For purpose of the experiment we remove the exploration portion of the UCB where we had

$$argmax_{c \in c(v,d)} \left(\frac{r(c)_{\rho(d)}}{n(c)} + k\sqrt{\frac{\log n'(c)}{n(c)}} \right)$$

We added,

$$argmax_{c \in c(v,d)} \left(\frac{r(c)_{\rho(d)}}{n(c)} \right)$$

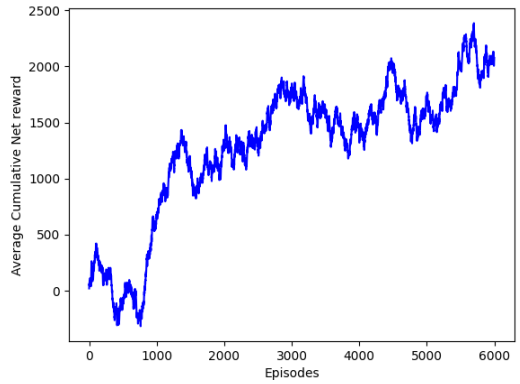
instead, in an effort to maximize rewarding branch non-zero visited branch. We ran this for 400 games and with a tree node size of 1000. In the results it was found due to the loss of exploration of non-zero visited node there was decline in win percentage against the random player to 45.33%.

Increasing the number of nodes For this part of the experiment we try increase the number of node to see if better performance can be yielded. Hence we increase the number of node to 10000 for 300 games to see if the win percentage increases.
Result, caused a win rate of 62% with a run time of 14hrs

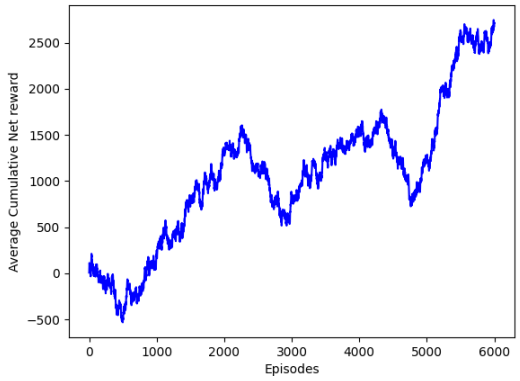
DDQN and DRQN

In this experiment, DDQN and DRQN player are playing against random and greedy agent for 5 trials of 6000 episodes. Rewards are difference between the player and opponent scores at the end of episode. Cumulative net rewards over episode is used to show learning curve. Initially, for first 500 episode no update are done on network. Hence, there is no learning in initial 500 episode which can be seen in below graphs. Therefore, net cumulative rewards is decreasing initially. But, after 500 episodes all 3 agents are able to learn and gain high rewards.

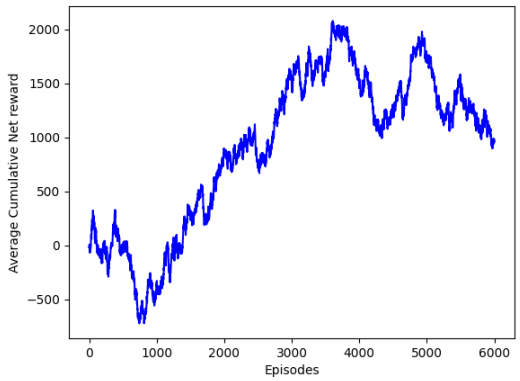
– Average of Cumulative net rewards gain by "Double DQN" after 5 trails of 6000 epsiodes against random agent.



– Average of Cumulative net rewards gain by ”DRQN” after 5 trails of 6000 epsiodes against random agent.



– Average of Cumulative net rewards gain by DRQN after 5 trails of 6000 epsiodes against greedy agent.



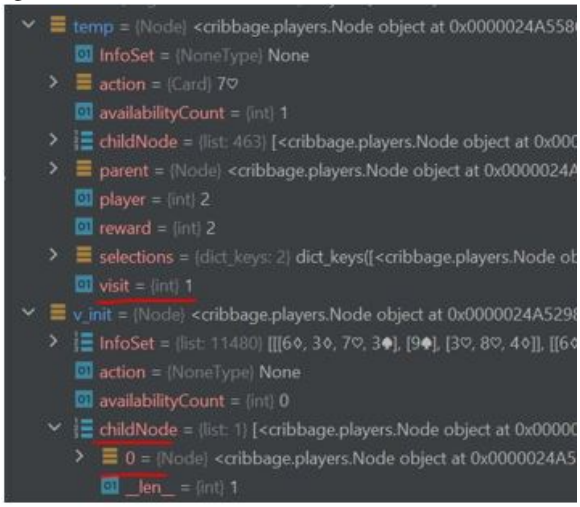
Similar, learning curve is seen for DDQN again greedy. Greedy is play with more accurracy resulting in large decrease in reward gained at initial stages, net rewards are going be -500. But, one agent has stated learning slow agent was able to learn pattern and beat greedy in long run. Thus, DDQN and DRQN algorithm was able to gain higher rewards against random and greedy. Initial, higher value of batch size(1000) is used on which learning was very less. Reducing the batch size to 128 has show more stable learning results.

Conclusion

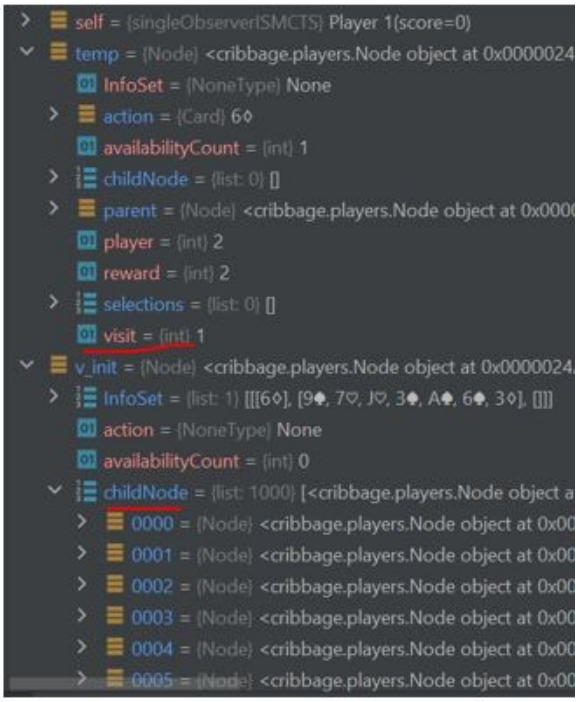
SO-ISMCTS

We have learned from the run that for SO-ISMCTS to perform optimally to give a high percentage of wins needs to a significantly large sum of node to effectively play a partially observable game like cribbage. The game itself has high amounts of random noise causing it not to have a high win rate. In the implementation we observed in the beginning the number possible games states to explore is too high as we have the entire enemy hand of any 4 possible cards out of 37 cards to explore which has 101,270 possible combinations but as soon as the number of enemy cards to explore went down due to the progression of the game easier it became for the algorithm to explore.

For example, here is a log picture of a 1000 node iteration at the start of the game this is when we have to pick a child node action which is at the end of the agents run:



Here ’v init’ contains childnodes actions to be chosen that has the most visits this is when the game is stating out and both players have 3 or more cards in their hand in contrast to this log where player have 1 card or no cards



Here on the other hand we have a high selection of childnode actions to choose from.

DDQN and DRQN

We have the learned that DDQN and DRQN has show good learning against random and greedy. But due stochastic of environment lot of fluctuation in the rewards are seen. More training of network might help to learn more stable policy in such non-deterministic environment. DRQN has shown slow learning, but learning is more stable with less fluctuations spikes as compare DDQN. Large number of training iteration and advance alogrithm like NFSP might help to learn better policy to play cribbage and come up with state of art performance.

References

[1] Silver, D., Guez, A., Hasselt, H. v. (2015, 12 8). [1509.06461] Deep Reinforcement Learning with Double Q-learning. arXiv. Retrieved October 26, 2022, from https://arxiv.org/abs/1509.06461

[2] Zha, Daochen and Lai, Kwei-Herng and Cao, Yuanpu and Huang, Songyi and Wei, Ruzhe and Guo, Junyu and Hu, Xia, RLCard: A Toolkit for Reinforcement Learning in Card Games, 2019. https://arxiv.org/abs/1910.04376

[3] biblatex @phdthesiskelly2017comparison, title=Comparison of Monte Carlo Tree Search Methods in the Imperfect Information Card Game Cribbage, author=Kelly, Richard and Churchill, David, year=2017, school=Memorial University

[4] biblatex @articlecowling2012information, title=Information set monte carlo tree search, author=Cowling, Peter I and Powley, Edward J and Whitehouse, Daniel, journal=IEEE Transactions on Computational Intelligence and AI in Games, volume=4, number=2, pages=120–143, year=2012, publisher=IEEE

[5] "Is Cribbage a Game of Luck or Skill? (Win More!) – Card Game King." Card Game King, https://cardgameking.com/cribbage-luck-vs-skill/. Accessed 12 December 2022