POKER ENGINE
AI PROJECT REPORT

PREPARED FOR

Dr. Poulami Dalapati
Assistant Professor
Department of Computer Science &
Engineering - LNMIIT, Jaipur

PREPARED BY

Rahul Agarwal 19UCS264
Vaibhav Kagathara 19UCS181
Manish Porwal 19UCS184
Akshat Chhaparwal 19UCS211
Pushkar Raj Upadhyay 19UCS219

Abstract

In recent years there has been a great rise in artificial intelligence (AI), with games often serving as challenge problems, benchmarks, and milestones for progress. Poker is a game with incomplete and imperfect information. The ability to estimate opponent and interpret its actions makes a player as a world class player. Making a poker game in artificial intelligence is not a new idea. However the inherent randomness in the game creates a challenge for developers and researchers all around the world and is catching a lot of attention. In this project we are developing a poker agent using evolutionary techniques and tested against other agents.

Table of Contents

Introduction to Poker Engine

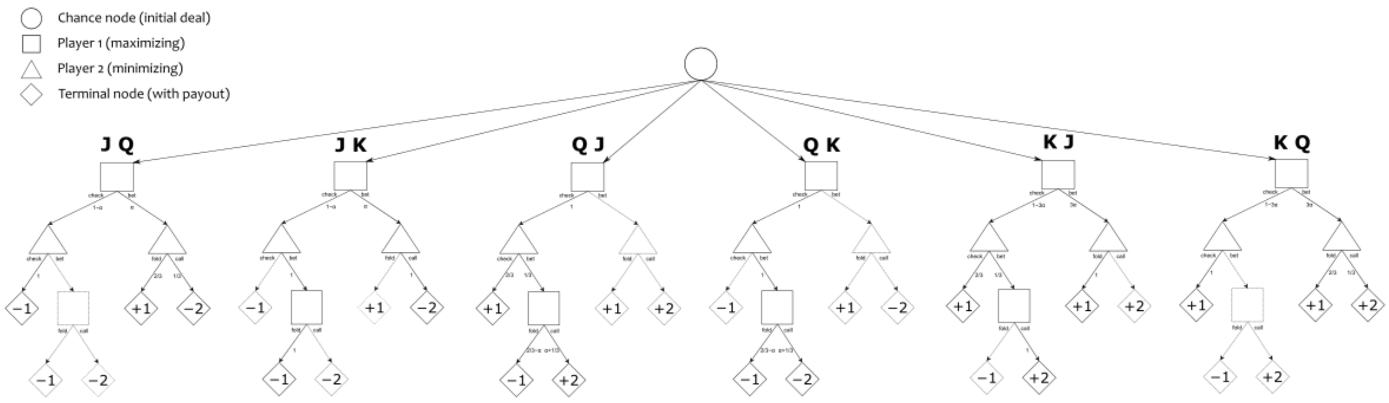
We have built an AI engine to play poker. The type of poker this engine can play is Kuhn poker. The deck of Kuhn poker consists of three playing cards, including a King, Queen, and Jack. Each player is given one card and is able to put bets in the same way as they would in a Texas Hold 'Em poker game. If both players bet or pass, the player with the higher card wins, else the betting player wins.

Understanding Kuhn Poker

The basics of Kuhn poker are as follows:

1. Kuhn Poker Defaults
- Initially at the starting of each round in the game each player antes 1.
- One of the three cards is handed to each participant, while the third is kept hidden aside.

2. Kuhn Poker Playing Conditions
- Player one can pass or bet 1.
    - If player one passes then player two can pass or bet 1.
        - If player two passes there is a showdown for the pot of 2 (i.e. the higher card wins 1 from the other player).
        - If player two bets then player one can pass or bet.
            - If player one passes then player two takes the pot of 3 (i.e. winning 1 from player 1).
            - If player one bets there is a showdown for the pot of 4 (i.e. the higher card wins 2 from the other player).
    - If player one bets then player two can pass or bet.
        - If player two passes then player one takes the pot of 3 (i.e. winning 1 from player 2).
        - If player two bets there is a showdown for the pot of 4 (i.e. the higher card wins 2 from the other player).



Input & Output of the Program

The input for the game is the card value (0 for ‘J’, 1 for ‘Q’, 2 for ’K’) for player 1, if the player 1 passes then for the next round we will require the actions of the opponent (0 for pass and 1 for bet).

The output will be whether the AI agent will pass or bet i.e. 0 or 1 respectively for each round if applicable.

Solving Real World Problem

While AI had some success at beating humans at other games such as chess and Go (games that follow predefined rules and aren't random), winning at poker proved to be more challenging because it requires strategy, intuition, and reasoning based on hidden information. Despite the challenges, artificial intelligence can now play and win poker.

Implementation Set-Up Environment

1. Hardware

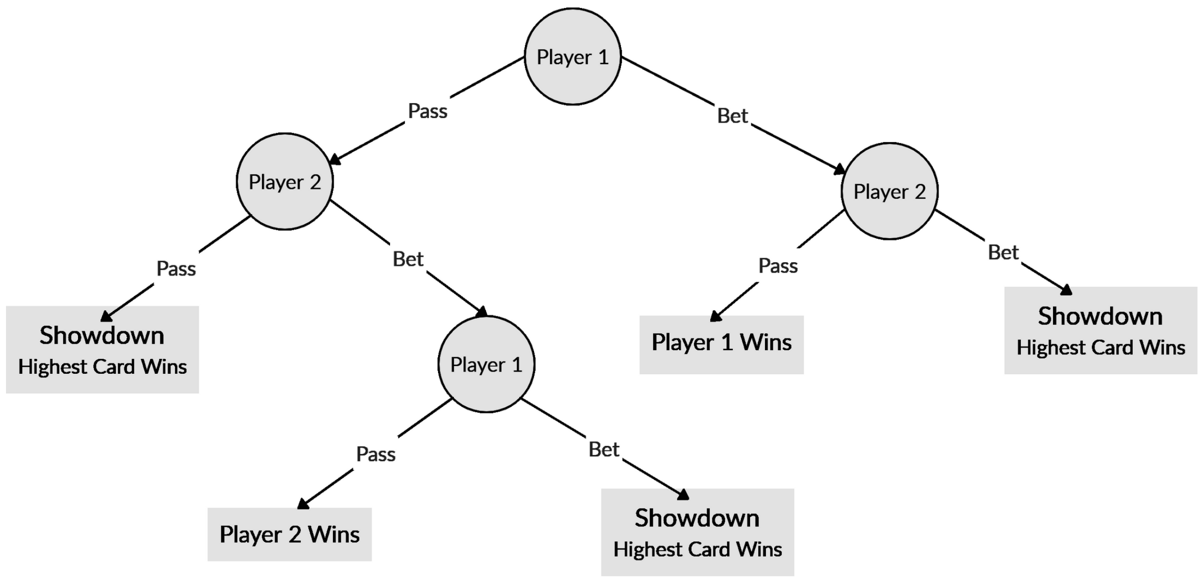
- Operating System : Windows 10 64 bit
- CPU : 64-bit Intel(R) Core(TM) i5-8300H Quad Core CPU @ 2.30GHz
- RAM : 8GB DDR4 Memory

2. **Software**

- Python Version : 3.10.0
- numpy Version : 1.21.4

Modeling the Task

Firstly, we are creating a game tree in which the root node represents the player1, left and right child of roots represent player2 and right of left child of root represent player1 and all the values of regrets and strategies are initialized to zero. We then trained the model and in each iteration the player1 and player2 are given 2 cards randomly shuffled between 3 cards and their learning process begins.



We calculated the actual reward which is the reward that the player1 will get at the end of the game and its negation will be the reward that player2 will get. After that we calculated regret both for player 1 and player2 by calculating counterfactual reward which is the reward that they could have got by making other decisions. The regret is now calculated by taking the difference between actual reward and counterfactual reward. The goal is to minimize the regret and gain maximum reward. After every iteration we normalized the regret sum and updated the probability of actions. In the training itself after every iteration the player would get biased according to the probability in order to minimize regret. Atlast we got our result as a matrix which basically tells what action to take(pass or bet) according to the probability and the state of the agent.

Challenges of building the system

The first and most important challenge in building this system was to analyze and model the possible scenarios of the two players in different states depending on what actions(pass or bet) they take.This was done by game tree implementation in which each node represented attributes of both players. After this, the second main challenge was to train our analyzed model and for that, we have used the Counterfactual regret minimization algorithm which is based on reinforcement learning technique.

How our Method address the challenges

In our model we have implemented a single game tree in which we have all the players in the gametree itself but in another method we have 2 different game trees each for different players. In the current model use of 1 gametree helps in computing and storing the values for regret and strategy in a better way which helps us to apply Counterfactual regret minimization algorithm in a single game tree without applying it in two game trees.

Pros
- By using a single gametree it helps us to reduce space complexity.
- It also helps us to reduce the cyclomatic complexity of the code.

Cons
- During the training we are calculating the player's actual reward and counterfactual reward separately by visiting the complete game tree twice which eventually increases the time complexity.

Appropriate Algorithms for handling the model

We used a domain specific algorithm which is counterfactual regret minimization and in games like rock-paper-scissors,tic-tac-toe,Poker etc which are partially observable in nature.

Implementation

To build this poker engine, we have implemented a game tree in python which we used to train our AI agent. We have created two classes:

(1) Node:
It has a name, regretSum and strategy as member variables. Here regretSum is a 3x2 matrix where rows represent card values and columns represent pass and bet regret for each node. Similarly we have a strategy as a 3x2 size matrix which stores strategy for each row and column. We have two functions namely:

- init(): it assigns name to the node and initializes regretSum and strategy matrix values to 0

- getProbability(): It gives the probability of pass and bet for that particular node and card by calculating the normalized regretSum. If the normalized regretSum is greater than 0 then we will calculate the probability by dividing regretSum with normalisedSum else we are using equal probability for pass and bet. After that we are adding the probability to the current strategy of the particular node and returning the probability.

- getStrategy(): It returns the probability of each strategy by dividing it with strategySum for that node and card.

(2) KuhnPoker:

- init(): it initializes a card array with values 0, 1 and 2 denoting ‘j’,‘q’,’k’ respectively.

- getActualReward(): In this function we are travelling the gameTree and choosing either pass or bet using the getProbability() function for that particular node. After reaching a leaf node we are returning the reward and the path that we have travelled according to the Kuhn poker rules.

- playerCounterfactReward(): In this we are adding regrets, in the regretSum matrix for nodes representing player1 by subtracting the actual reward of player1 with the counterfactual reward.

- opponentCounterfactReward(): In this we are adding regrets, in regretSum matrix for nodes representing player2 by subtracting the actual reward of player2 with the counterfactual reward.

- train(): For each iteration we are first shuffling the cards and giving two cards one to player1 and other to player2. Then we are calculating the actual reward for player1 by calling getActualReward() and assigning negative reward to player2. Then we are calling playerCounterfactReward() and opponentCounterfactReward() to calculate regrets for nodes and cards.