ARTIFICIAL INTELLIGENCE

POKER ENGINE

Under the guidance of :

Dr. Poulami Dalapati

Dr. Nitin Kumar

Members:

19UCC023 - Mohit Akhouri

19UCC026 - Divyansh Rastogi

19UCC119 - Abhinav Raj Upman

19UCS024 - Hardik Satish Bhati

19UCS064 - Tapomay Singh Khatri

Department of Computer Science and Engineering

The LNM Institute of Information Technology, Jaipur

LNMIIT
The LNM Institute of Information Technology

TABLE OF CONTENTS

INTRODUCTION

PROBLEM STATEMENT :

Build an engine to play poker online.

DESCRIPTION :

In this project , we were able to build a simplified version of the poker game - which is "Limit Texas Hold'em Poker". Some rules of the poker one should keep in mind before trying to build a poker engine are as follows :

- While playing poker , there is a **fixed amount** of money - nothing more or less coming in or going out.
- **Uncertainty** of the cards in possession of your opponent and uncertainty of the community cards on the table.

After carefully analysing the elements and rules of the poker game. We have come up with some analysis about the game. They are:

- Poker is a **zero-sum** game with an element of uncertainty.
- It is not computationally feasible to look at all the possible states of a poker game.
- We need to think about a smarter approach to make better decisions regarding which move to make. We need to think about the concept of **heuristics** to solve this problem.

OUR AIM FOR THE PROJECT :

After carefully analysing the rules of the poker game and through some of the conclusions we obtained after the analysis, our team have come up with an aim to design and implement an **optimal** poker engine to play a 2-player game of “Limit Texas Hold’em Poker”.

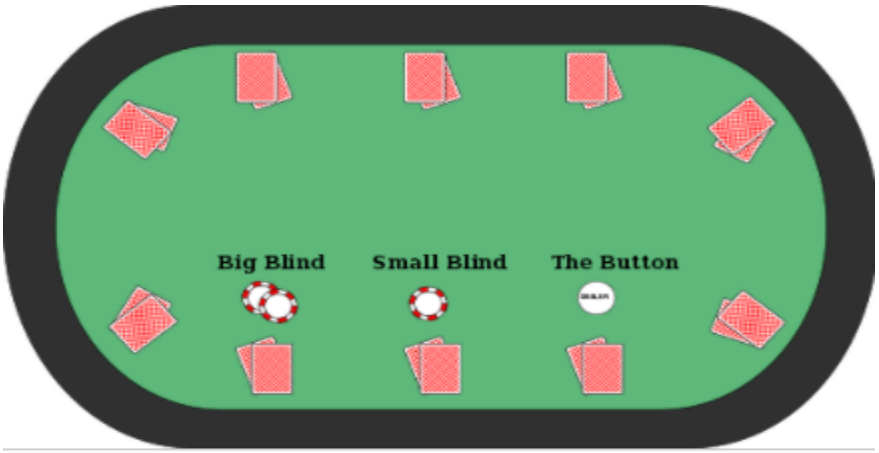
ABOUT LIMIT TEXAS HOLD'EM POKER

Texas hold'em is one of the most popular variants of the card game of poker. There are two types of cards namely :

- **Hole cards :** These cards are dealt face down to each player.
- **Community cards :** These cards are dealt face up in three stages

The stages consist of series of three cards (" the flop"), later an additional single card ("the turn" or "fourth street") and a final card ("the river" or "fifth street"). Each player seeks the **best five** card poker from any possible combination of seven cards. Players have betting options to **check**,**call**,**raise** or **fold**. Rounds of betting take place before the flop is dealt and after each subsequent deal. The player who has the best hand and has not yet folded by the end of all betting rounds wins all of the money bet for the hand, known as the **pot**. In certain situations, a **split-pot** or **tie** can occur when the two players have hands of equivalent value. This is also called a **chop-pot**.

The skills required to play this game can be probability , psychology , game theory and strategy.



The above figure shows the position of the blinds relative to the dealer button. The above structure of the poker table is during the initial **betting stage** of the game.

OUR APPROACH

Monte Carlo method :

Monte Carlo methods, or Monte Carlo experiments, are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle. They are often used in physical and mathematical problems and are most useful when it is difficult or impossible to use other approaches.

In principle, Monte Carlo methods can be used to solve any problem having some probabilistic interpretation. By the law of large numbers, integrals described by the expected value of some random variable can be approximated by taking the empirical mean (a.k.a. the sample mean) of independent samples of the variable. When the probability distribution of the variable is parametrized, mathematicians often use a Markov chain Monte Carlo (MCMC) sampler.] The central idea is to design a judicious Markov chain model with a prescribed stationary probability distribution. That is, in the limit, the samples being generated by the MCMC method will be samples from the desired (target) distribution.By the ergodic theorem, the stationary distribution is approximated by the empirical measures of the random states of the MCMC sampler.

In the case of a poker engine we can use Monte Carlo Tree Search.

Monte Carlo Tree Search or short MCTS is a simulation based search algorithm that can be used to solve extensive-form games with imperfect information. Complex games like NLTH have a large number of states due to the element of chance and the number of opponents. Instead of searching the complete state space (which is intractable) MCTS only samples one possible outcome per iteration. As the number of iterations increases, the true value of a state can be approximated. MCTS is a anytime algorithm which means that the computation can be interrupted at any time to retrieve a valid result (or an approximation of it if not enough iterations have been performed.) The search is performed on a tree containing the following types of nodes:

Root node is the start of the tree. The node represents the state we want to evaluate.

Action nodes are nodes in which the same player has to make a decision as in the root node.

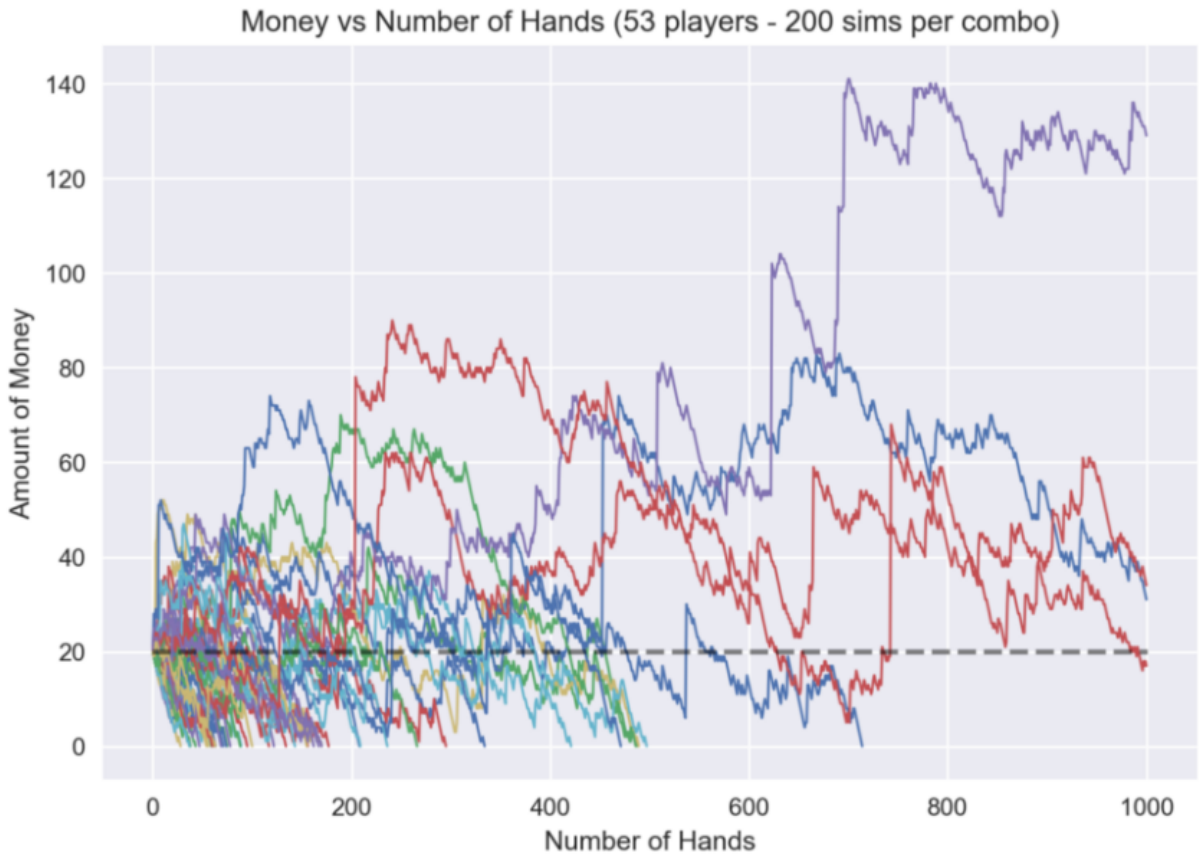
Opponent nodes represent the opposing player's decisions.

Terminal nodes represent either nodes in which the game ended by folding by all but one player or by going to a showdown.

Chance nodes are used in places where chance makes a move (draw a card).

Each iteration begins at the root node and recursively selects children according to a selection strategy that chooses which action to take. If the state for the selected action does not exist, the last state is expanded and it's children are added to the tree before continuing the selection until a terminal history is reached. This history is then evaluated according to a simulation strategy that calculates the expected value. In the final step of the iteration the calculated value is propagated back up the tree by a backpropagation strategy. After a certain amount of time or number of predefined iterations a final move strategy selects the actual actions to play from the root node.

An example of monte carlo simulation run on the game of poker is as follows :



CODE

Setup Python and Installation of packages :

Open a terminal , create a new directory named as **pokerengine**.

```
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~$ mkdir pokerengine
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~$ cd pokerengine
```

Next we need an engine through which we can simulate our poker engine(bot).To do that first, we need to install the **PyPokerEngine** package using the command **pip :**

```
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~/pokerengine$ pip install pypokerengine
Collecting pypokerengine
  Downloading PyPokerEngine-1.0.1.tar.gz (44 kB)
     |████████████████████████████████| 44 kB 2.0 MB/s
Building wheels for collected packages: pypokerengine
  Building wheel for pypokerengine (setup.py) ... done
  Created wheel for pypokerengine: filename=PyPokerEngine-1.0.1-py3-none-any.whl size=34035 sha256=7a0561a8467a0044e40386e990faaafd86b7e206155f6ecd4f8533f0158fffbc
  Stored in directory: /home/jovyan/.cache/pip/wheels/dc/3c/c0/2e6817e2d991c09b03834c439204d172bfcbfd94f02464e945
Successfully built pypokerengine
Installing collected packages: pypokerengine
Successfully installed pypokerengine-1.0.1
```

PyPokerEngine is a simple framework for **Texas hold'em AI development.**

Installation of GUI for graphically displaying the poker game :

We can install the following package **pypokergui** for graphical representation of the game.

```
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~/pokerengine$ pip install pypokergui
```

Here we have chosen **pypokerengine** and **pypokergui** for simulation of Limit Texas Hold'em poker.

Implementation of Poker Engine :

We have created three files for the simulation of the poker engine. We have implemented the poker engine in **PYTHON** through the use of package **pypokerengine**. The description of the different files are as follows :

(1) poker_engine_main_file.py code :

This is the main file for the implementation of poker engine. It employs **Monte Carlo Simulations** for the working of poker engine. It estimates the win rate and also calculates the **optimal amount** obtained after playing with the poker bot.

Code for importing of packages is as follows :

```
# AI PROJECT - POKER ENGINE
# poker_engine_main_file.py

# 19ucc023 = Mohit Akhouri
# 19ucc026 = Divyansh Rastogi
# 19ucc119 = Abhinav Raj Upman
# 19ucs024 = Hardik Satish Bhati
# 19ucs064 = Tapomay Singh Khatri

# importing of packages from pypokerengine starts here
from pypokerengine.engine.hand_evaluator import HandEvaluator
from pypokerengine.players import BasePokerPlayer
from pypokerengine.utils.card_utils import _pick_unused_card, _fill_community_card, gen_cards
```

We have imported the packages required for the working of the poker engine. First package that we required is **HandEvaluator** from pypokerengine.engine. It evaluates the value of cards in the hands of the player or bot. We also import the package **BasePokerPlayer** from pypokerengine.players. We imported various types of packages like **_pick_unused_card , _fill_community_card and gen_cards.**

Code for the function win_rate_estimation :

```python
# ALGORITHM FOR win_rate_estimation function is as follows :
# This function will estimate the ratio of winning games corresponding to the current state of the game
def win_rate_estimation(nb_simulation, nb_player, hole_card, community_card=None):
    # nb_sim = simulation variable
    # nb_player = to store the value of player
    # hole_card = to store the value of hole card
    # community_card = to store the value of community card
    if not community_card: community_card = []

    # Making lists of card objects and storing them in two variable out of a deck of cards
    community_card = gen_cards(community_card) # separating community cards from the deck
    hole_card = gen_cards(hole_card) # separating hole cards from the deck

    # to estimate the win count , perform Monte Carlo simulation
    wincount = sum([montecarlo_sim(nb_player, hole_card, community_card) for _ in range(nb_simulation)])
    wincount_modified = 1.0 * wincount / nb_simulation
    return wincount_modified
```

This function is designed to estimate the **win_rate** for a player. First the community card **c_card is** initialized. Next we will do the separation of **community** and **hole** cards from the deck of cards. We have used the function **gen_cards**. Lastly, we will estimate the win rate using Monte Carlo Simulation. We also calculate the **modified** win rate and return that win rate.

Code for the function montecarlo_sim :

```python
# ALGORITHM FOR montecarlo_sim function is as follows :
# This function will perform montecarlo simulation on nb_player,h_card and c_card and return the values 0 and 1
# the function will return 1 if player_score > bot_score
def montecarlo_sim(nb_player, hole_card, community_card):
    # Do a Monte Carlo simulation given the current state of the game by evaluating the hands

    # performing monte carlo simulation according to the current state of the game
    # the method applied will be evaluation of hands
    community_card = _fill_community_card(community_card, used_card=hole_card + community_card)
    unused_cards = _pick_unused_card((nb_player - 1) * 2, hole_card + community_card) # picking up unused cards

    # calculation of holes and score of bot and player
    bot_hole = [unused_cards[2 * i:2 * i + 2] for i in range(nb_player - 1)]
    bot_score = [HandEvaluator.eval_hand(hole, community_card) for hole in bot_hole]
    player_score = HandEvaluator.eval_hand(hole_card, community_card)

    # checking condition = if player_score > bot_score , return 1 else 0
    if player_score >= max(bot_score):
        return 1

    return 0
```

This function is designed for the implementation of monte carlo simulations. We have performed simulations according to the current state of the game. We also calculated the holes and score of both bot and player. Lastly, the condition is checked that if player_score > bot_score , we will return 1 else 0.

Code for the declaration and initialization of poker_engine_main_file class :

```
# class file for poker_engine_main_file
class poker_engine_main_file(BasePokerPlayer):

    # initializing wins and losses
    def __init__(self):
        super().__init__()
        self.wins = 0
        self.losses = 0

    # creating functions for actions taken in a game of poker
    def declare_action(self, valid_actions, hole_card, round_state):

        # hole_card = to store the value of hole card
        # round_state = variable to store round state in game of poker

        # estimation of win rate
        win_rate = win_rate_estimation(100, self.num_players, hole_card, round_state['community_card'])

        # checking the condition of 'call' state in poker
        can_call = len([item for item in valid_actions if item['action'] == 'call']) > 0

        # if condition satisfied we will calculate the call amount else assign call amount value 0
        if can_call:
            # If so, compute the amount that needs to be called
            call_amount = [item for item in valid_actions if item['action'] == 'call'][0]['amount']
        else:
            call_amount = 0
```

This code contains the details of the **class poker_engine_main_file** , first we declare a __init__(self) function to initialize the number of wins and losses. A function called **declare_action** is created that will declare all the valid actions a player/bot can take in a poker game. This function also calculates the win rate and checks the condition of the call state for assigning the proper value to variable **call_state_amount** , basically it calculates the **optimal amount**.

Code for the calculation of amount according to the win rate :

```
        amount = None # variable to store the amount

        # If we find that win rate is very large , then we should go for 'raise' state
        if win_rate > 0.5:
            raise_amount_options = [item for item in valid_actions if item['action'] == 'raise'][0]['amount']
            if win_rate > 0.85:
                # Raise the amount as much as possible if there is extremely likely chance to win
                action = 'raise'
                amount = raise_amount_options['max']
            elif win_rate > 0.75:
                # Raise by minimum amount if there is a likely chance to win
                action = 'raise'
                amount = raise_amount_options['min']
            else:
                # If we find there is 'chance' to win , then we should go for 'call' state
                action = 'call'
        else:
            action = 'call' if can_call and call_amount == 0 else 'fold'

        # Setting the amount variable
        if amount is None:
            items = [item for item in valid_actions if item['action'] == action]
            amount = items[0]['amount']

        return action, amount
```

Code for the declaration of inherited functions from player.py :

```
    # receive message regarding start of poker game
    def receive_game_start_message(self, game_info):
        self.num_players = game_info['player_num']

    # receive message regarding start of round state of poker game
    def receive_round_start_message(self, round_count, hole_card, seats):
        pass

    # receive message regarding start of street state of poker game
    def receive_street_start_message(self, street, round_state):
        pass

    # receive message regarding updation of game
    def receive_game_update_message(self, action, round_state):
        pass

    # receive message regarding result of round state of poker game
    def receive_round_result_message(self, winners, hand_info, round_state):
        is_winner = self.uuid in [item['uuid'] for item in winners]
        self.wins += int(is_winner)
        self.losses += int(not is_winner)

# defining setup function for poker_engine_main_file.py
def setup_ai():
    return poker_engine_main_file()
```

This code will **inherit** functions like **receive_game_start_message** for receiving of message on start of poker game. **Receive_round_start_message** is used for receiving messages regarding round start. We also inherit functions like **receive_street_start_message** for receiving message regarding start of street round , **receive_game_upate_message** to receive message regarding update of game. Lastly we have inherited the function **receive_round_result_message** which provides the message regarding results of poker rounds.

We have also defined a function **setup_ai** which will return the object corresponding to **poker_engine_main_file**.

(2) CallBot.py code :

Code for the calling the Poker engine :

```
# CallBot.py python file
# This code will call the poker engine for simulation

# Importing of packages - pypokerengine , numpy and sklearn
from pypokerengine.players import BasePokerPlayer
import numpy as np
from sklearn.neural_network import MLPRegressor

class CallBot(BasePokerPlayer):
    def declare_action(self, valid_actions, hole_card, round_state):
        actions = [item for item in valid_actions if item['action'] in ['call']]
        return list(np.random.choice(actions).values())

    # Inheriting functions from players.py python file
    def receive_game_start_message(self, game_info):
        pass

    def receive_round_start_message(self, round_count, hole_card, seats):
        pass

    def receive_street_start_message(self, street, round_state):
        pass

    def receive_game_update_message(self, action, round_state):
        pass

    def receive_round_result_message(self, winners, hand_info, round_state):
        pass

# Definition of function setup_ai for creating an object of CallBot class and then return it
def setup_ai():
    return CallBot()
```

The definition of various functions that we have declared or inherited from players.py are as follows :

declare_action: This function is called, when the player is asked for an action. We will have to provide your action and an amount. In this bot, we simulate 1000 games to see how many we will win. If the probability of winning the game is greater than 1/num_players (10% for 10 players), we will call. Otherwise we fold. This way, we only play the good hands. Note that the fold-action is in place 0 in the valid_actions array, call/check is in place 1 and raise is in place 2.
receive_game_start_message:This function is called when a new game begins. Hence is only called once per game, and the game_info dictionary contains numerous game info, like the number of players.
receive_round_start_message: This function is called preflop when a new round starts. Here, we have the round count, the cards on your hand (holecard) and information about the other players in the game.
receive_street_start_message: This function is called when a new “street event” happens, that is pre-flop, at flop when community cards are shown, and when turn and river are shown.
receive_game_update_message: This function is Called each time a player performs an action
receive_round_result_message: This function is called when a round is over. This happens either after a showdown or after all players fold. When the implementation is done for more advanced bots, we can use this method for updating the bots.

(3) simulate.py code :

Code for the simulation of Limit Texas hold'em poker engine :

```
# simulate.py python file
# This code will call the pypokerengine, CallBot and poker_engine_main_file
# for simulation of Limit Texas Hold'em poker

# importing required packages
from pypokerengine.api.game import start_poker, setup_config
from CallBot import CallBot
from poker_engine_main_file import poker_engine_main_file
import numpy as np

# code for computation of stack log of poker bot starts here
if __name__ == '__main__':
    poker_bot = poker_engine_main_file() # initializing object of poker_bot

    # The stack log contains the total amount in stacks of the poker bot after each game (the initial stack is 100)
    stack_log = []

    # running a loop for each round for calculation of stack amount , total rounds = 100
    for round in range(100):
        p1, p2 = poker_bot, CallBot()

        # configuration setup and registration of players for the game
        config = setup_config(max_round=5, initial_stack=100, small_blind_amount=5)
        config.register_player(name="p1", algorithm=p1)
        config.register_player(name="p2", algorithm=p2)
        # calculation of poker game result
        game_result = start_poker(config, verbose=0)

        # appending new stack values each time
        stack_log.append([player['stack'] for player in game_result['players'] if player['uuid'] == poker_bot.uuid])
        print('Avg. value in stack :', '%d' % (int(np.mean(stack_log))))
```

The bot uses Monte Carlo simulations running from a given state. Suppose the player starts with 2 high cards (two Kings for example), then the chances that the player will win are high. The Monte Carlo simulation then simulates a given number of games from that point and evaluates which percentage of games the player will win given these cards. If, during the flop another King shows, then the player's chance of winning will increase. The Monte Carlo simulation starting at that point, will yield a higher winning probability as the player will win more games on average.

After running the simulations, we can see that the bot based on Monte Carlo simulations outperforms the always calling bot. If the player starts with a stack of $100,-, the player will on average end with a stack of $120,- (when playing against the always-calling bot).

OBSERVATIONS

Run the following command in python terminal for obtaining the results :

```
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~/pokerengine$ python simulate.py
```

After running the commands for 100 iterations(rounds of poker game), we will obtain the **amount in the stack** of poker_bot for each round of the game.

```
jovyan@jupyter-ipython-2dipython-2din-2ddepth-2d416sparr:~/pokerengine$ python simulate.py
Avg. value in stack : 80
Avg. value in stack : 90
Avg. value in stack : 126
Avg. value in stack : 108
Avg. value in stack : 102
Avg. value in stack : 96
Avg. value in stack : 100
Avg. value in stack : 87
Avg. value in stack : 95
Avg. value in stack : 95
Avg. value in stack : 102
Avg. value in stack : 110
Avg. value in stack : 117
Avg. value in stack : 112
Avg. value in stack : 109
Avg. value in stack : 106
Avg. value in stack : 108
Avg. value in stack : 111
Avg. value in stack : 116
Avg. value in stack : 113
Avg. value in stack : 112
Avg. value in stack : 116
Avg. value in stack : 119
Avg. value in stack : 115
```

Analysis of the results obtained :

We have implemented the **Monte Carlo Simulation** in the design of poker games. Initially, we have taken the stack amount to be 100. The code was run for 100 rounds and obtained the stack amount for each round. After carefully analyzing the results, we realised that our poker agent has a **high chance** of winning if rounds of the game are shorter. Now if we increase the number of rounds in the game, there is a high chance that the human player will beat the poker agent.
So the final conclusion is that Monte Carlo Simulation was **efficient** in the design of the poker engine. We can also employ various other algorithms like **adversarial approach** , **Reinforcement learning with neural networks** in the designing of poker bot. Some of the references to these algorithms we have provided in the references section of this report.

CONCLUSION

We came up with a poker agent that can play poker against other agents with different strategies and win or come really close to winning the bigger pot of money. Our agent has a high chance of winning short games. From the above Monte Carlo Simulation we concluded that Monte Carlo simulations Using built in function estimate hole card win rate. We analysed the results of the poker game by using Monte Carlo Simulations. The Monte Carlo simulations provided by the PokerEngine turned out to be efficient. We can also go for various other algorithms like reinforcement learning , adversarial search and neural networks for the design of poker game.

ACKNOWLEDGEMENTS

We express our sincere thanks to our professors Dr. Poulami Dalapati and Dr. Nitin Kumar for giving us the opportunity to design a project based on a Poker engine. We are overwhelmed and indebted to everyone who has helped us develop our ideas and researching regarding this topic. We express our gratitiude to our parents who are an inspiration for us. This project has helped us in increasing our skills and knowledge.

REFERENCES

Building poker agent using reinforcement learning with neural networks - Annija Rupeneite
[https://www.scitepress.org/Papers/2014/51489/51489.pdf]

Pypoker engine Github - 10 December , 2020
[https://github.com/epifab/pypoker]

Texas Hold'em Poker bot in python - Andreas Thiele , 23 June , 2018
[https://medium.com/@andreasthiele/building-your-own-no-limit-texas-holdem-poker-bot-in-python]

About Monte Carlo Simulation - 12 November 2021
[https://en.wikipedia.org/wiki/Monte_Carlo_method **]**

What is Monte Carlo Simulation - Metis , 12 February 2018
[https://medium.com/@thisismetis/what-is-a-monte-carlo-simulation-part-3-a9c1b68f8e6e]

About Limit Texas hold'em poker - 7 November 2021
[https://en.wikipedia.org/wiki/Texas_hold_%27em]

Docs of pypoker
[https://ishikota.github.io/PyPokerEngine/]