Modelling a DRL agent to play the game of Rock Paper and Scissors

1st Surya Teja Chavali, 2nd Rama Sai Abhishek Podila, 3rd Charan Tej Kandavalli
, 4th Amudha J.
Department of Computer Science and Engineering, Amrita School of Computing, Bengaluru, Amrita Vishwa Vidyapeetham, India.
bl.en.u4aie19014@bl.students.amrita.edu, bl.en.u4aie19053@bl.students.amrita.edu,
bl.en.u4aie19013@bl.students.amrita.edu, j_amudha@blr.amrita.edu

***Abstract*—Rock, Paper, Scissors (RPS) is an intelligent game when it comes to the sequence of strategic moves being able to predict the next move of the opponent and win. The traditional approaches only involve deciding the type of move RPS given the user's move as input. But building a strategy to win against the opponent in a sequence of plays cannot be implemented using such approaches. When using a DRL-based tactic, we prefer to see evidence that the AI agent can adapt to the opponent's shifting tactics and keep outperforming them in terms of win rates. In traditional approaches, the agent is static, which means that the agent always wins in every case due to the already given user's input. But using DRL, we can achieve an equal chance of winning or losing as the agent learns in real-time. Exploiting such non-random behavior tasks and recognizing patterns in opponent's play to win the game is a challenging and interesting problem to solve. A sort of semi-supervised learning employing incentives as labels is used in this work, in contrast to previous RL projects where the AI agent has to learn a task effectively and generalise to other future data. However, in this specific design, there isn't any conclusion to the game, meaning that it never ends. The RL agent essentially keeps evolving and adjusting as best it can.**

***Index Terms*—RPS, Deep Reinforcement Learning (DRL), Pattern Recognition**

I. Introduction

Despite the fact that a significant percentage of deep reinforcement learning (DRL) research has concentrated on video game applications and simulated control, they have very less implementation connecting the dots with respect to the limitations of learning in actual real-world situations. The necessity for intelligent machines to modify their behaviour in response to other agents' activities is a typical demand. The gaming business, where it is essential to generate players capable of adapting to the tactics of each opponent, places a specific emphasis on such conduct because it makes the game more challenging and realistic. In a broader sense, strategy games—whether they be electronic or not—tend to offer favourable conditions for the investigation and creation of such agents. Rock-paper-scissors is a particularly interesting game because of how straightforward it is and how it may be compared to circumstances where agents are not given a vast range of action options. A number of communities and sub populations have satisfactorily been represented using this game analogy.

II. Literature Survey

Very few notable works have been published in this sector of DRL.

The authors in [6] used the rock-paper-scissors game to study the nature of the learning process in monkeys engaged in a competitive game with ternary options. Each animal demonstrated preferences for some targets under the initial scenario where the computer chose its targets at random. They investigated mechanisms which were better represented by a model that combined the characteristics of both models, as opposed to straightforward models of reinforcement learning or belief learning. These findings imply that stochastic decision-making tactics used by monkeys in social interactions may be modified in response to both real and potential rewards.

Later, Tengfei Ma et al. have investigated whether the rock-paper-scissors (RPS) imaging task has pattern distinctions in the brain in [6]; their research aims to categorise this task using fNIRS and deep learning.They also employed a total of 22 people for the trial and created an RPS assignment that lasted 25 minutes and 40 seconds. Furthermore, they used the fNIRS acquisition device (FOIRE-3000) to record the cerebral neural activities of these participants in the RPS task. The time series classification (TSC) algorithm was introduced into the time-domain fNIRS signal classification. Experiments show that CNN-based TSC methods can achieve 97% accuracy in RPS classification.

A much more light-weighted model is proposed in [7], where Muhammad Nur Ichsan et al. have conducted and implemented a study to enable a more accurate picture classification model than those used in earlier research in order to discriminate between the stone, paper, and scissors hand movements. Consequently, Convolutional Neural Network, a Deep Learning technique, is applied in this classification procedure (CNN). The CNN model has been improved by this research by incorporating epoch values and more detailed hyper parameters.

The above mentioned research works are on either simple RPS game winning models/agents using Q - learning or analyzing the pattern of moves played, but without the use of state-of-art deep reinforcement learning techniques. This work aims to design and build a Double DQN agent which can accurately learn the strategy/sequence of moves played by the opponent using greedy policy in the game of RPS.

III. Methodologies

The AI agent serving as Player 1 was built using the double-DQN RL method in this project. Remember that Watkins invented Q-Learning in 1992; Deep Mind played a significant role in its current popularisation and has made numerous architectural improvements. The reference section contains a few well-known studies on DQNs (deep Q-learning networks). Learning a ”policy” (which is actually a deep learning network) that optimizes the anticipated return of future action is the essence of deep learning. The ”action value” function is the Q-value. Although its absolute size is meaningless, it is derived repeatedly to instruct the system on the best course of action to follow at any given time, with the goal of maximizing potential benefits in the future

A. Double Q-Learning

The issue of significant over estimations of action value (Q-value) in basic Q-Learning was addressed by the proposal of double Q-learning. The optimum course of action to take in each situation is always the Agent’s optimal strategy in basic Q-learning. The underlying premise of the proposition is that the optimal action has the highest expected/estimated Q-value. The Agent must first estimate Q(s, a) and update them after each iteration because it has no prior knowledge of the surroundings. Such Q-values include a great deal of noise, and we can never be certain that the course of action with the highest expected or estimated Q-value is actually the optimal one. Unfortunately, the Q-values of the best action is typically lower than those of the non-optimal ones. The Agent tends to choose the less-than-optimal action in any given state simply because it has the highest Q-value, according to the best practice in basic Q-Learning. The term for this issue is **”over estimations of action value”** (Q-value). The noises from the predicted Q-value will result in significant positive biases in the updating process when such an issue arises. As a result, learning will be an extremely difficult and untidy process.

The following are some enhancements used in this design that goes above and beyond standard Q-learning. Deep Mind has introduced the majority of these in recent years:

- Dual models (thus the word ”double”): one model serves as the target model, and the other serves as the model that drives the action decision. In Every move cycle, the inner weights are ”transferred” on a discounted basis from the action model to the target model.
- Using the epsilon-greedy technique, exploitation versus exploration This is crucial in any RL design because exploration is necessary to look for potential improvements in the optimal operation position by looking in various areas of the state space, particularly at the beginning of the process. Exploration is frequently referred to as ”off-policy,” whereas acting in accordance with exploitation.
- Experience replay: Deep Mind proposed the idea of experience replay, in which previous iterations of the state space are preserved in memory rather than using the most recent history as the learning space. These memories are employed in the SGD process and recalled (sampled) at each move (thus achieving the reinforcement notion).

Over estimation plays a role because, if the noises of all Q-values have uniform distribution, more specifically, Q-values are equally overestimated, then over estimations are not the problem since these noises don’t impact on the difference between the *Q(s’, a) and Q(s, a).*

B. Why Double Q-learning?

- By breaking down the maximum operation in the target into action selection and action evaluation, Double Q-learning aims to reduce over estimations.
- The target network in the DQN design offers a natural choice for the second value function, although not being completely decoupled, without the need to construct additional networks.
- Therefore, we suggest assessing the greedy policy using the online network while estimating its value using the target network.

Since the present Q (s, a), which is quite noisy, is used to estimate the best Q (s, a). In the Loss Function, the difference between the best and current Q (s, a) is similarly clumsy and contains positive biases brought on by various disturbances. These favorable biases have a significant impact on the update process. **This is exactly why positive biases happen.**

$$LossFunction = Q_{best}(s,a) - Q(s_t, a_t) \tag{1}$$

C. Double Deep Q Network: Experience Replay

We store the agent’s experiences at each time step ’t’ using the reinforcement learning replay memory technique called ”experience replay”, pooled over many episodes into a replay memory.

This means that the system saves the data discovered for [state, action, reward, next state] - often in a huge database - rather than executing Q-learning on state/action pairings as they arise during simulation or real-world experience. Notably, associated values are not stored here; rather, this is the raw data that will be used to calculate future action-values. The process of learning is then logically distinguished from that of gaining experience and is based on selecting samples at random from this table. Because changing the policy would result in different behaviour that should explore actions closer to optimal ones, you still want to interleave the two processes of acting and learning because you want to learn from them. You can divide this up as you choose; for example, take one step and learn from three unrelated preceding steps. There is no different formula for the Q-Learning targets when using experience replay because they are the same targets as in the online version. You would also use the loss formula provided for DQN without experience replay. Only the s, a, r, s’, and a’ that you feed into it makes a difference.

$$E(t) = s(t), a(t), r(t), s(t+1) \tag{2}$$

D. *Advantages And Disadvantages: Experience Replay*

By building on prior knowledge, it can be used more effectively. This is crucial so that you may get the most out of your real-world experience, which can be expensive. Since Q-learning adjustments are incremental and take time to converge, it is preferable to perform multiple runs on the same data, especially when there is no variation in the short - term solutions (reward, next state) for the same state and action pair. When training a function approximator, better convergence behavior. This is partially due to the fact that the data is more similar to the i.i.d. data that most supervised learning convergence proofs presuppose. Utilizing multi-step learning algorithms, such as Q (), that can be adjusted to produce superior learning curves by balancing bias (caused by bootstrapping) and variance, is more difficult.

IV. System Architecture

A. *DRL Model Design*

Agent - The 3d-printed bot which plays Rock, Paper, Scissors

Environment - The opponent who is playing against the agent (input is given as the sequence of pattern in the moves).

- Our aim is to make the agent learn through multiple failed attempts and try to mimic the behaviour of the user and defeat him over a period of time.
- To obtain this , we can implement using Deep Q Networks in Reinforcement Learning.

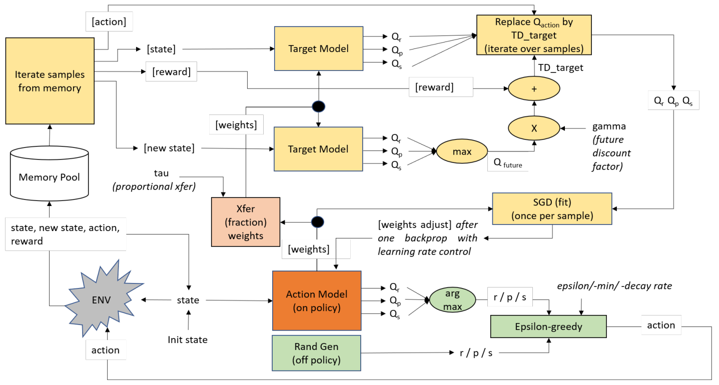


Fig. 1. Architecture of a DQN Model

B. *Markov Decision Process (MDP) Modelling*

The interaction contains the following:

- Action space: The AI agent (player 1) throws out a game of scissors, paper or rock.
- Rewards: A signal coming from the outside world to player 1. If player 1 wins, the payoff is simply a value of 1, and if not, a value of 0 when its lost, and 0 if the state is a tie.
- State: Here's where things become interesting and creative. We have created the state space in this configuration to be: Each of the three outcomes—a victory, a draw, or a loss—can be set to 1 for a specific state.
- WinRateTrend, TieRateTrend, LostRateTrend: These metrics shows if a moving average with a positive trend (set to 1) is present or otherwise (set to 0). The three indications are each evaluated separately.
- WinRateMovingAvg, TieRateMovingAvg, LostRateMovingAvg: This decimal value, which ranges from 0 to 1, represents the rate. This rate is determined using a rolling average window size that has been set, to how far is the agent learning in window of some number of episodes for all three types.

C. *Double Deep Q Network*

Deep Q Network (DQN) and Target Network, two distinct Deep Neural Networks, are used in Double DQN. Since it will be applied during the optimization stage of updating the Deep Q Network parameters, it should be noted that there is no learning rate when updating the Q-values.

$$Q_{qnet}(s_t, a_t) \leftarrow R_{t+1} + \gamma(Q_{tnet}(s_{t+1}, a)) \tag{3}$$

$$a = max_a Q_{qnet}(s_{t+1}, a) \tag{4}$$

$$q_{estimated} = Q_{tnet}(s_{tnet}, a) \tag{5}$$

Summarizing this architectural design, the basic goal of the game is to see whether the RL agent can adapt to the changing behaviours and keep up a high victory rate. Since the RPS game is fundamentally a time series (sequence) problem, a straightforward full mesh DNN turns out to be one of the primary limitations, which is further detailed in sub - section D.

V. Training

A set of training data is used to teach a deep reinforcement learning agent how to make predictions or execute actions in a certain environment. The agent "learns" from the feedback provided and modifies its internal parameters as needed to enhance performance over time. In order to train the deep Q-network (DDQN), we give it a sizable dataset of such sequences, along with the accompanying rewards for making accurate predictions. This would allow you to train the DDQN to predict the next move in a rock-paper-scissors series. In order to optimize the rewards it receives for producing accurate predictions, the DDQN would then apply reinforcement learning to modify its internal settings. Iteratively, the DDQN can be trained on ever bigger and more intricate datasets to enhance its performance.

We define the problem that the agent is supposed to solve before we train a DDQN agent to find patterns in sequences of moves in the game of rock, paper, scissors. In this scenario, the agent's objective is to anticipate a human player's subsequent move based on the previous moves that have been made in the game. The DDQN agent must be trained on a sizable dataset of game sequences and their accompanying upcoming moves in order to achieve this goal. This dataset can be created by simulating rock, paper, scissors games between two players or it can be gathered from games that actual people play in the real world.

We start the process of training the DDQN agent after the training data (which is an input sequence) has been gathered. The definition of the DDQN's architecture, which includes the number and kind of layers, the dimensions of the input and output layers, and any other hyper parameters that can have an impact on the network's performance, is the initial stage in this procedure.

After the DDQN's architecture has been established, we may start training the network using the gathered dataset. In order to reduce the error between the anticipated future moves and the actual next moves in the training dataset, the network's weights and biases are adjusted using an optimization procedure, such as stochastic gradient descent.

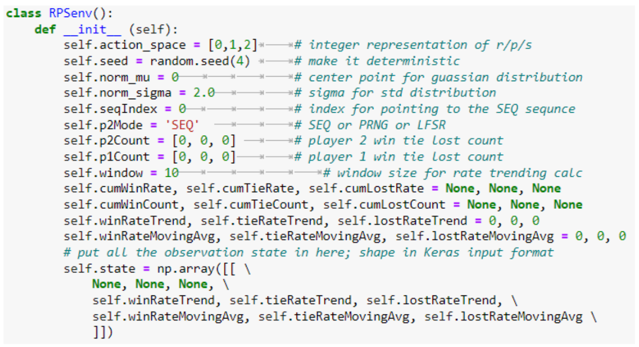
A. Hyper Parameters

The following are a group of game hyper - parameters that can change the AI agent's overall flexibility and victory rate:

- **Memory batch size** - The convergence would occur more quickly but adaptability would be slower with a larger batch since more reinforcement will be used on each episode. Additionally, the RAM maximum length is included. Python's deque object collection is used for this. Faster convergence may result from experience retention brought on by longer memories, but adaptability may suffer since it takes time to clear the memory when a opponent player's behaviour changes.
- **Gamma** - Gamma helps you choose what portion of the developing action-value pair you would like the algorithm to take into account at each iteration. This impacts an episode's overall return value, which helps us gain deeper understanding.
- **DNN Layers** - The entropy or the randomness of NN and its capacity to learn the spread are effectively represented by the DNN Layers.
- **Reward System** - 1 for win, -1 lose and 0 for draw: Utilizing simple rewards and allowing the complexity of adaptation to be managed via state space and NN architectures.
- **Epsilon** - The exploration % and decay rate are both under the direction of Epsilon. It's critical to notice that there is a minimum value because the system will keep looking at off-policy movements because that is how it picks up on new opponents' tendencies.

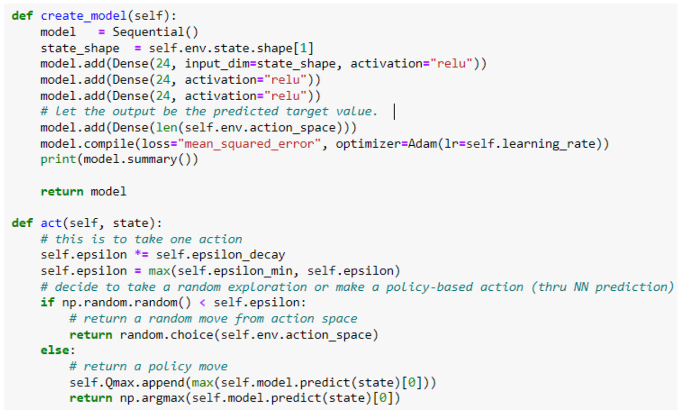
VI. Demo Code

Below are some sample images of the Custome environment ans the DDQN agent classes.



```
class RPSenv():
    def __init__ (self):
        self.action_space = [0,1,2]     # integer representation of r/p/s
        self.seed = random.seed(4)      # make it deterministic
        self.norm_mu = 0                # center point for guassian distribution
        self.norm_sigma = 2.0           # sigma for std distribution
        self.seqIndex = 0               # index for pointing to the SEQ sequnce
        self.p2Mode = 'SEQ'             # SEQ or PRNG or LFSR
        self.p2Count = [0, 0, 0]        # player 2 win tie lost count
        self.p1Count = [0, 0, 0]        # player 1 win tie lost count
        self.window = 10                    # window size for rate trending calc
        self.cumWinRate, self.cumTieRate, self.cumLostRate = None, None, None
        self.cumWinCount, self.cumTieCount, self.cumLostCount = None, None, None
        self.winRateTrend, self.tieRateTrend, self.lostRateTrend = 0, 0, 0
        self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg = 0, 0, 0
        # put all the observation state in here; shape in Keras input format
        self.state = np.array([[ \
            None, None, None, \
            self.winRateTrend, self.tieRateTrend, self.lostRateTrend, \
            self.winRateMovingAvg, self.tieRateMovingAvg, self.lostRateMovingAvg \
            ]])
```

Fig. 2. Rock Paper Scissors Custom built Environment



```
def create_model(self):
    model   = Sequential()
    state_shape  = self.env.state.shape[1]
    model.add(Dense(24, input_dim=state_shape, activation="relu"))
    model.add(Dense(24, activation="relu"))
    model.add(Dense(24, activation="relu"))
    # let the output be the predicted target value.  |
    model.add(Dense(len(self.env.action_space)))
    model.compile(loss="mean_squared_error", optimizer=Adam(lr=self.learning_rate))
    print(model.summary())

    return model

def act(self, state):
    # this is to take one action
    self.epsilon *= self.epsilon_decay
    self.epsilon = max(self.epsilon_min, self.epsilon)
    # decide to take a random exploration or make a policy-based action (thru NN prediction)
    if np.random.random() < self.epsilon:
        # return a random move from action space
        return random.choice(self.env.action_space)
    else:
        # return a policy move
        self.Qmax.append(max(self.model.predict(state)[0]))
        return np.argmax(self.model.predict(state)[0])
```

Fig. 3. Small sample of the DDQN Agent

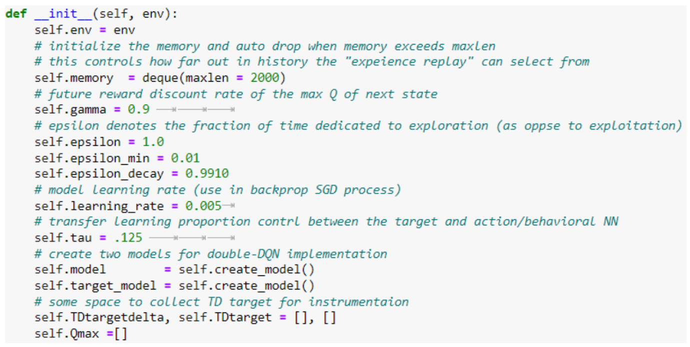
VII. Results and Discussion

The entire execution this work has taken place in Jupyter Notebook with Python (version - 3.7.9), on a machine running on a Intel core i7 processor, a RAM of 16 GB and a GPU equipped with NVIDIA GeForce GTX 750.

2 different experiments with parameter tuning have been performed, through we analyze and conclude some inferred results as follows:

Experiment - 1:

- Gamma = 0.7
- Epsilon = 1.0
- Epsilon_Minimum = 0.01
- Epsilon_Decay = 0.9910



```
def __init__(self, env):
    self.env = env
    # initialize the memory and auto drop when memory exceeds maxlen
    # this controls how far out in history the "expeience replay" can select from
    self.memory  = deque(maxlen = 2000)
    # future reward discount rate of the max Q of next state
    self.gamma = 0.9
    # epsilon denotes the fraction of time dedicated to exploration (as oppse to exploitation)
    self.epsilon = 1.0
    self.epsilon_min = 0.01
    self.epsilon_decay = 0.9910
    # model learning rate (use in backprop SGD process)
    self.learning_rate = 0.005
    # transfer learning proportion contrl between the target and action/behavioral NN
    self.tau = .125
    # create two models for double-DQN implementation
    self.model        = self.create_model()
    self.target_model = self.create_model()
    # some space to collect TD target for instrumentaion
    self.TDtargetdelta, self.TDtarget = [], []
    self.Qmax =[]
```

Fig. 4. Agent Hyper - Parameters

- Learning Rate = 0.005
- Tau = 0.125

Experiment - 2:

- Gamma = 0.9
- Epsilon = 1.0
- Epsilon_Minimum = 0.01
- Epsilon_Decay = 0.884
- Learning Rate = 0.01
- Tau = 0.5

Inference: When calculating the goal Q values, Gamma controls how much future benefits are discounted. It is a scalar value between 0 and 1 that controls how much weight is given to anticipated future rewards when calculating goal Q levels. Gamma can train the algorithm to give long-term incentives a higher priority than short-term ones by increasing the weight that future benefits are given in the calculation. Therefore, in the former experiment, since the gamma value is considered to be 0.7, the rewards aren't much getting discounted, thus an increment of 0.2 made the return values for each episode higher, resulting in greater overall scores.

The algorithm's level of exploration versus exploitation is determined by epsilon. It is a scalar value between 0 and 1, and instead of utilising the model to predict the action that will maximise the expected reward, it estimates the likelihood of executing a random action in a given state. A larger epsilon number indicates that the algorithm is more likely to rely on the model's predictions, whereas a lower value indicates that the algorithm is more likely to execute random exploratory activities.

The rate at which the value of epsilon depreciates over time is determined by decay. It is a scalar value between 0 and 1, and at each step, it is multiplied by the current value of epsilon to reduce its value. This gives the algorithm a mechanism to successfully balance exploration and exploitation, which can aid in its ability to learn.

The speed at which model weights are transmitted to the target model is controlled by tau. The weight of the new weights from the model in the weighted average used to update the weights in the target model is determined by this scalar value, which ranges from 0 to 1. The target model's weights will be changed more quickly to keep up with the model's weights when tau is larger, which can increase learning process stability and performance.

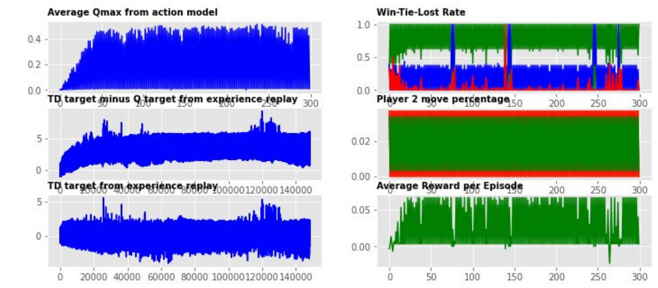


Fig. 5. Model Evaluation Before Hyper Parameter Tuning

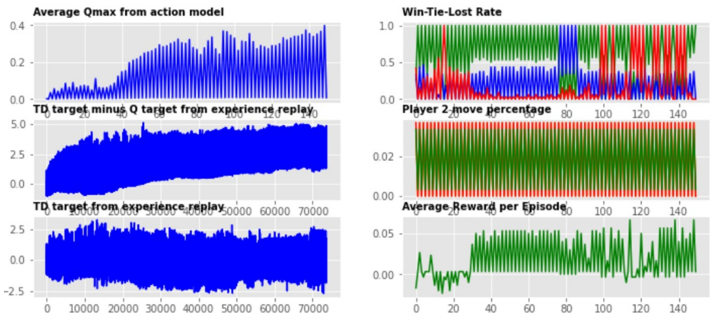


Fig. 6. Performance post parameter tuning

References

[1] Souza, Diego & Carneiro, Hugo & França, Felipe & Lima, Priscila. (2013). Rock-paper-scissors WiSARD. Proceedings - 1st BRICS Countries Congress on Computational Intelligence, BRICS-CCI 2013. 10.1109/BRICS-CCI-CBIC.2013.38.

[2] H. Brock, J. Ponce Chulani, L. Merino, D. Szapiro and R. Gomez, "Developing a Lightweight Rock-Paper-Scissors Framework for Human-Robot Collaborative Gaming," in IEEE Access, vol. 8, pp. 202958-202968, 2020, doi: 10.1109/ACCESS.2020.3033550.

[3] A. Castro-González, J. C. Castillo, F. Alonso-Martín, O. V. Olortegui-Ortega, V. González-Pacheco, M. Malfaz, et al., "The effects of an impolite vs. a polite robot playing rock-paper-scissors" in Social Robotics, Cham, Switzerland:Springer, pp. 306-316, 2016.

[4] K. Ito, T. Sueishi, Y. Yamakawa and M. Ishikawa, "Tracking and recognition of a human hand in dynamic motion for janken (rock-paper-scissors) robot", Proc. IEEE Int. Conf. Autom. Sci. Eng. (CASE), pp. 891-896, Aug. 2016.

[5] https://www.cs.utexas.edu/ pstone/Courses/394Rfall16/resources/9.5.pdf

[6] D. Lee, B. P. McGreevy, and D. J. Barraclough, "Learning and decision making in monkeys during a rock-paper-scissors game," Brain Res. Cogn. Brain Res., vol. 25, no. 2, pp. 416–430, 2005.

[7] T. Ma, W. Chen, X. Li, Y. Xia, X. Zhu, and S. He, "fNIRS Signal Classification Based on Deep Learning in Rock-Paper-Scissors Imagery Task," Applied Sciences, vol. 11, no. 11, p. 4922, May 2021, doi: 10.3390/app11114922.

[8] Muhammad Nur Ichsan, Nur Armita, Agus Eko Minarno, Fauzi Dwi Setiawan Sumadi, and Hariyady, "Increased Accuracy on Image Classification of Game Rock Paper Scissors using CNN", J. RESTI (Rekayasa Sist. Teknol. Inf.) , vol. 6, no. 4, pp. 606 - 611, Aug. 2022.