Project Report
on

An Independent Study of Reinforcement Learning for Games

In partial fulfillment of the requirements
for the award of the degree
Of

MASTER OF COMPUTER SCIENCE

By
Harshvardhan Rajpurohit
200485741

Department of Graduate studies
University of Regina
3737 Wascana Pkwy,
Regina, SK S4S 0A2

August, 2024

Abstract

Reinforcement learning (RL) is a machine learning paradigm where agents learn optimal behaviors through trial and error. They receive rewards or penalties for actions, adjusting their strategies to maximize cumulative rewards. RL involves the exploration of new actions and exploitation of known ones, guided by algorithms like Q-learning and deep RL. One such algorithm is SARSA (State-Action-Reward-State-Action), an on-policy RL algorithm that updates the action-value function based on the action taken by the agent, considering the current state, the chosen action, the received reward, the next state, and the next action. This project uses SARSA to implement a drawing-card game where the agent draws a specific number of cards from a deck and aims to maximize the combined value of the drawn cards.

Contents

Table of Figures

Glossary

Below are some commonly used terms in the report:

1. *Agent*: The learner or decision maker that interacts with the environment, understands it, and performs an action. The environment rewards or punishes the agent based on its actions.

2. *Environment*: Everything the agent interacts with and reacts to. In other words, is the external system, task, simulation, or all the elements that an agent interacts with, such as a gaming environment.

3. *State*: A representation of the current situation in the environment.

4. *Action*: A set of all possible steps or moves the agent can make in the environment.

5. *Reward*: Feedback from the environment to evaluate the effectiveness of an action.

6. *Policy*: A strategy that the agent uses to determine the next action based on the current state. Refers to a series of actions that the agent follows in a time step after observing the environment.

7. *Value Function*: This component assesses or predicts the value of being in a state and calculates the cumulative reward the agent will be able to collect if it performs a particular action. Estimates how good a state or state-action pair is in terms of future rewards.

8. *Dictionary*: A data structure in Python that stores key-value pairs. Each key is unique and acts as an identifier for its associated value, enabling efficient data retrieval.

Dictionaries are used to organize and manage data by mapping keys to their corresponding values.

9. *List*: A data structure in Python that holds an ordered collection of items, which can be of different data types. Lists are mutable, allowing for dynamic modification of their content. They are commonly used to store sequences of data elements and support operations like indexing, slicing, and iteration.

Introduction

Reinforcement learning is a type of machine learning algorithm that involves training an agent, by interacting with the environment, to make decisions maximising a numerical reward over a period of time. Agents interact with the environment and make decisions which yields a numerical reward, based on which the value of the reward is calculated. A bad decision yields a negative reward while a good one yields positive reward, thus highlighting the influence of the actions on the environment. By following this trial-and-error approach of making some interactions and receiving rewards, agent tend to learn about the nature of the environment and over time, tend to make use of this learnings. Initially, when agents are unaware of the environment, they tend to explore it by pursuing every action possible, after a while when enough data is gathered, agents tend to look back at the learnings and take the actions yielding good rewards provided a similar scenario occurred in the past, this will be the exploitation of environment. The strategy that the agent follows to interact with the environment is termed to be as policy. Based on this policy, there are additional distinctions that tend to yield different outcomes. These distinctions are on-policy and off-policy methods. On-policy methods, such as SARSA (State-Action-Reward-State-Action), follow a strategy to learn about the environment and then later use the same policy when evaluating the best possible actions to achieve the maximum outcome. They tend to update the strategy based on their learnings and then later use for evaluation. Off-policy methods, such as Q-learning, tend to follow a different approach. These methods tend to follow a different strategy for learning about the environment but does not have the constraint of using the same strategy for evaluation. These can have a different strategy for

evaluating the actions while a whole different strategy for choosing the appropriate actions based on the states. They can utilize data generated from other policies or even random actions, allowing for more aggressive exploration and potentially faster learning.

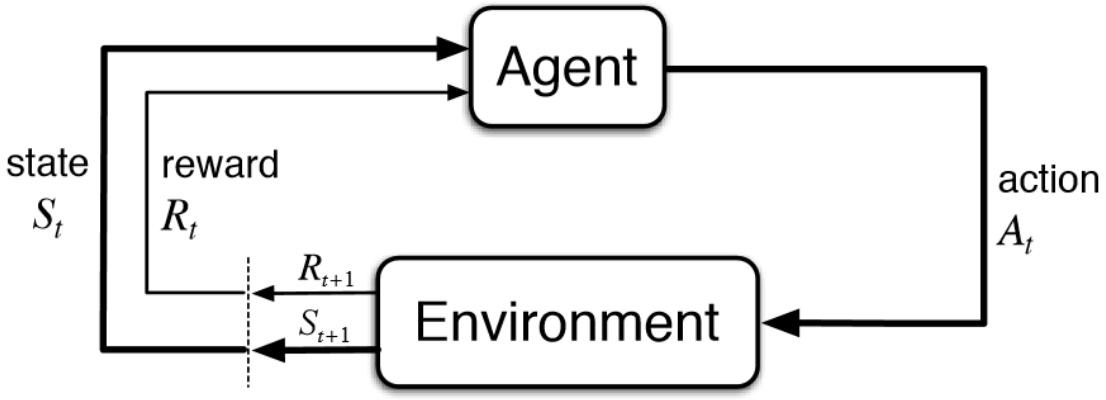

Figure 1: General framework of Reinforcement Learning

The main goals of this project are to comprehend the principles of reinforcement learning (RL) and to put these concepts into practice by creating an RL algorithm in a simple card game scenario. The project aims to investigate how RL agents learn the best strategies by interacting with an environment, with the main focus on maximizing the value of cards in hand. The scope of the project involves studying important RL methods, with a particular focus on on-policy algorithms like SARSA, and understanding how they can be applied in finite, discrete state spaces such as card games. The project also includes creating a simulation of a card drawing game where the agent learns to maximize the cumulative value of non-repetitive value cards through trial and error. By the end of the project, a fully functional RL model will be implemented, demonstrating the practical application of theoretical RL concepts in a simplified gaming context. This will provide insights into the

decision-making processes and learning dynamics of RL agents. SARSA is a good choice for the project since its on-policy method updates the policy based on the actions taken, effectively handling exploration-exploitation trade-off. With a finite state and action space and conservative updates, SARSA ensures stable and reliable learning to optimize the cumulative reward in the form of card values in the game.

In the forthcoming sections of the report, we will provide detailed explanations of the project. The background section will extensively cover Reinforcement Learning and the SARSA algorithm, offering in-depth details to establish a clear understanding of the code explanation. The Project methodology and overview will explain all the aspects of the project. It will commence with an explanation of the development approach, followed by an overview of the various processes, and conclude with an in-depth explanation of the functioning of the project's code components. Lastly, the conclusion section will dive into the enhancements to the current system and its features.

Background

Reinforcement Learning (RL) is much different compared to the other types of machine learnings, for it does not require learning labeled datasets or recognizing patterns, instead it follows the strategy of letting the system figure out the sequence of actions that yields the highest reward through trial and error. Since actions, in some cases, not only impacts the immediate reward but also the future situations and the rewards received at those times. Therefore, the understanding of the how to take actions that would yield the best outcomes in the current environment, plays the

most important role in understanding reinforcement learning. RL is majorly influenced by how humans make decisions in real life. We tend to observe our actions, scenarios and situations and tend to learn from them. We incorporate these learnings in all the actions that follows the learnings. For instance, when someone is learning to cook, they try different recipes and cooking methods. The taste of the food acts as the results of the action. It tells them if they're doing a good job or not. This helps them get better at cooking over time since they start incorporating changes into the recipes and cooking things as per their own liking. Similarly, an agent in a RL environment observes the effects of its actions, based on the rewards or punishments it receives in return, and changes things so as to get the best outcome possible.

Following up on the cook example mentioned above, a recipe book seems to be the source of detailed instructions and actions, that needs to be performed during cooking. It also provides additional details about the expected results such that the cook can observe and compare the results of their actions, thus getting a clear understating about everything. Similar to this, a policy defines the behaviour of the agent learning the environment. Mapping from perceived environmental states to corresponding actions is what a policy is. Simply put, the policy is just a mapping of known environmental states to their corresponding actions. A policy can be as simple as a function or lookup table, or as complex as a search process but still plays an important role for RL agent since it determines the behavior of the agent all by itself. Similar to how some steps from the cookbook can be skipped, replaced with other steps or done in a different order, RL algorithms can also be defined to either follow the policy or go off of it. Methods or algorithms that strictly adhere to a single policy

and tend to change it accordingly during evaluation are termed to be as On-policy methods or algorithms, example SARSA (State-Action-Reward-State-Action). This means that they learn the value of the policy that is being executed currently thus resulting in more conservative updates, ensuring the learning is aligned with the agent's current behavior. Off-policy methods or algorithms, like Q-learning, use a different policy for learning the environment but do not tend to stick to the same policy when evaluation the actions of the agent. They have the ability to make use of data produced by other policies or even random actions, enabling more extensive exploration and potentially quicker learning. Off-policy methods can be more flexible and efficient but require careful management to ensure stability and convergence.

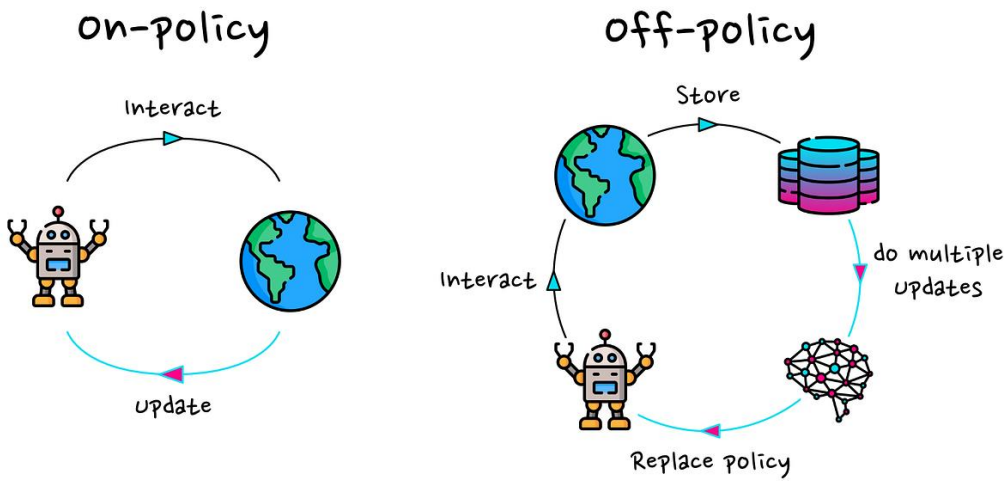

Figure 2: On-Policy vs Off-Policy Reinforcement learning

SARSA, is an on-policy algorithm that solely depends upon the current policy. The steps followed in the algorithm are mentioned in the name itself. SARSA stands for State-Action-Reward-State-Action, which is basically the major steps followed in the algorithm. The

process starts when the agent observes the current state (S) and chooses an action (A) based on its policy. After taking the action, the agent receives a reward (R) and observes the new state (S). Then it chooses the next action (A) based on its policy for the new state. SARSA updates the value of the current state-action pair using a formula that includes the reward received and the estimated value of the new state-action pair. This update rule ensures that the value function is adjusted to reflect the agent's actual experience, leading to a policy that is consistently refined based on the actions taken and their outcomes. Through this iterative process, SARSA gradually improves its policy to maximize cumulative rewards over time. This functioning of SARSA registers it as the perfect choice for the current project since its on-policy method approach updates the current policy in use thus rendering it appropriate for the final evaluation and referencing when used on the environment or a different one with same features.

Project Methodology and Overview

The following section provides a comprehensive explanation of the coding aspect of the project. It commences with a detailed explanation of the approach adopted, followed by an overview of the code structure, and concludes with an in-depth explanation of the individual code components.

Development Approach

The development approach for this project follows CLEAN architecture principles, ensuring a modular and maintainable codebase. The SARSA file implements the SARSA algorithm with methods for environment setup (`create_environment`), action selection (`choose_action`), value updates (`update`), training (`train`), and Q-value visualization (`display_Q_matrix`). The Q-learning file mirrors this structure to accommodate the Q-learning algorithm, with similar functionalities tailored to its principles. The Custom_environment file defines the game environment using the GYM library, providing a framework for interaction with the RL agents. Finally, the main file integrates these components, managing the overall process and enabling the agent to play the game by utilizing the RL algorithms and custom environment. This modular design enhances flexibility, facilitates easy updates, and maintains a clear organization throughout the project.

Implementation Details

The project implementation requires some key prerequisites to be clarified before we delve into the details. The most critical aspect for the project implementation is how the cards are represented. In this game, the cards are represented as a list of numbers ranging from 1 to 10 with no duplicate cards allowed. Essentially, it is a list of numbers from 1 to a number less than or equal to 10.

Given that reinforcement learning operates on state-action pairs, defining the states and actions is another crucial task. The actions for this simple card game involve discarding the drawn card from the deck (DISCARD), choosing the drawn card from the deck (PICK), or swapping the drawn card from the deck with one of the selected cards having the lowest value (SWAP). The states are structured to indicate the selected cards and the card drawn from the deck, referred to as the current card. The states are represented in the format (current card | selected cards). Each action within this particular state will impact either the current card or the selected cards, leading to a different state. All possible states are initially calculated and provided to the agent during training. The inputs from the user are defined in the main file, which contains of specifying the cards in the deck, essentially a list of numbers starting from 1 until any number equal to or less than 10, and a limit on how many cards can be selected by the agent.

Now, delving into a detailed explanation of the underlying code, we see that the project follows a structured approach consisting of three main steps. Firstly, a custom environment is created, then the agent is trained on this environment using a specific reinforcement learning algorithm, and finally, the trained agent is set to play the game with a modified environment. To effectively implement these steps, the entire project is divided into three separate files, each containing their own set of functions that collectively bring the project to life.

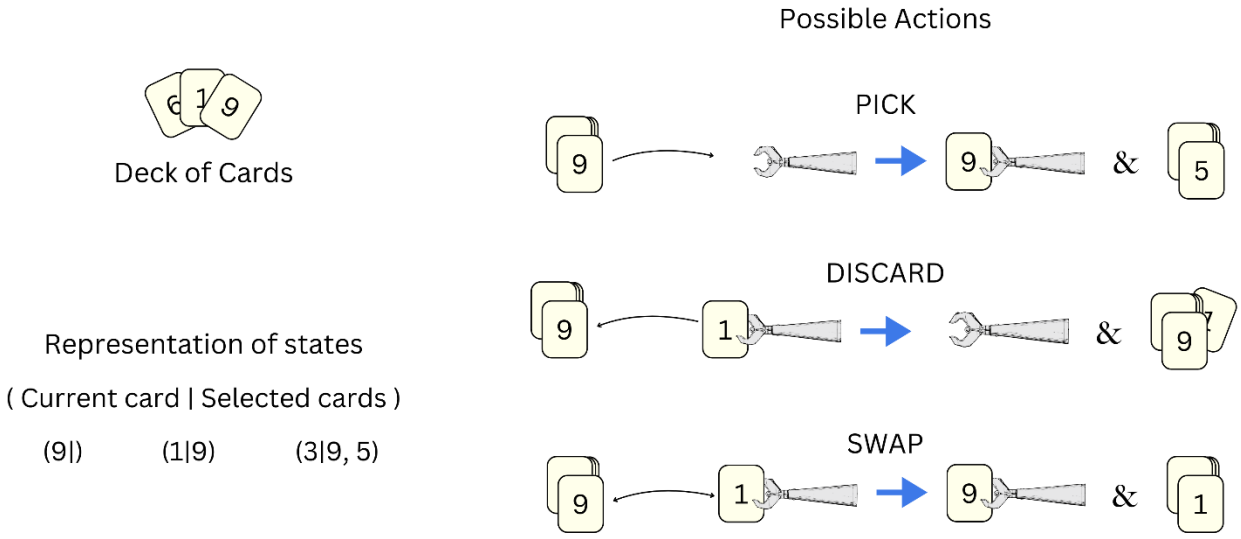

Figure 3: Representation of states, cards and possible actions

The *custom_environment.py* file is aptly named as it contains all the necessary code to create a customized environment using the gymnasium library. This file encompasses functions that establish the foundation for the game, enabling it to be used in training the agent during its training phases. The `SARSA.py` file incorporates functions that implement the reinforcement learning algorithm using the functions from the previous file. It takes responsibility for initializing the environment, training the agent, and populating the Q-table. Lastly, the `main.py` file serves as the starting point of the project, leveraging the other files and training the agent within an environment based on the input parameters. Additionally, it includes a function to deploy the trained agent to play the game with a randomized environment, which appears to be a shuffled deck for this case. The main file reads in the inputs provided by the user and traverses it to the other files when calling their respective function.

File defining Custom Environment (custom_environment.py)

The specified file contains all the required functions to define the environment in which the agent will undergo training and participate in the game. The process begins with the initialization of the environment, which is carried out in the `init` function of the class specified in the file. The primary objective of the `init` function is to create the observation space, which will be utilized for selecting states during both training and gameplay. During initialization, the first crucial step is to generate the observation space by providing it with inputs received from the main file. The `generate_observation_space` function takes these inputs and utilizes the permutations functionality from the numpy library to produce every possible state based on the provided inputs. It generates states in the same format as described previously and stores them in a list called states. The pseudocode for the function is provided below:

Initialize an empty **string variable** & an empty dictionary called **states_tuple_dict**.
Loop through the possible number of card selections:
- Generate all permutations of the deck for each selection size.
- Convert these permutations from tuples to strings and add them to the **string variable**
- Add these converted string permutations to **states_tuple_dict** as well

Initialize an empty list called **states**.
Split the combined permutations string into individual **states**
for each of these individual **states**:
- Convert each state in the state representation format
- Add to the **states** list

Return the list of formatted **states**

Figure 4: Pseudo code of function generating observation space

Once the list of all possible states is obtained, we proceed to generate every conceivable state by executing any action from the action space. This entails iterating through each state in the list and applying the actions from the action space, thereby creating new states. Subsequently, we establish a Python dictionary to document the relationships between the states. As each potential state stemming from the inputs has already been acquired, it is only necessary to link it to the outcome of a potential action on a specific state.

Initially, we classify the states based on their size, which is determined by the combined number of cards in the current card section and the selected card section. The rationale behind this classification is to evaluate the states based on their sizes and identify potential states for all actions. States of the same size can always be produced through the DISCARD or SWAP actions applied to any given state, while the PICK action inevitably alters the size of the state.

The method to differentiate between the DISCARD and SWAP actions hinges on the contents of the current card and selected card sections. When comparing any two states, if the current card section of both states differs while the selected card sections are the same, this signifies a DISCARD action in which a current card was discarded. Conversely, if the current card section and selected card sections of both states are distinct, and the value in the current card section has been swapped with one of the values from the selected card section, this indicates the performance of a SWAP action. This functionality is being performed in the `generate_possible_next_states_dict` function, whose pseudo code is provided below:

Initialize an empty dictionary called **possible_states_dict**
for each index in **states_tuple_dict**:
- Extract current set of states from **states_tuple_dict**
- Format these set of states in state representation format
- Append them into a list called **current_states**

- *if* the next set of states exists:
  - Extract set of states from **states_tuple_dict**
  - Format these set of states in state representation format
  - Append them into a list called **next_states**

for each state in **current_states**:
- Initialize a dictionary of possible actions (discard, swap, pick) for the state
- Extract **lhs**(current card) and **rhs**(selected cards) for the state

- *for* each state in the **current_states** except current state:
  - Extract **lhs**(current card) and **rhs**(selected cards) for the state
  - *if* **lhs** differs from current state's **lhs** and **rhs** is the same:
    - Associate it with DISCARD action for current state in **possible_states_dict**
  - *if* card swap is possible:
    - Associate it with PICK action for current state in **possible_states_dict**

- *for* each state in the **next_states**:
  - *if* **rhs** matches the current state's **rhs:**
    - Associate it with SWAP action for current state in **possible_states_dict**
  - *if* swapping cards is valid, update possible actions for current state in **possible_states_dict**

Return **possible_states_dict**

Figure 5: Pseudo code of function generating possible states

This file serves multiple purposes. First, it initializes the environment. Additionally, it contains two important functions: `step` and `reset`. The `step` function is responsible for executing the logic that enables the agent to take actions within the environment. When the agent performs an action, it considers the current state and uses a dictionary to determine the potential state resulting from the action. From the list of potential states, one is

randomly selected, and its associated reward value is calculated. This calculation involves comparing specific sections of the chosen state. The `step` function also validates the actions performed on the states to ensure consistency and marks the completion of training or playing episodes.

On the other hand, the `reset` function serves the purpose of resetting the current state to initiate a new episode of training for the agent. It resets the variables used for representing the current state and randomly selects a state from the list of available states. The pseudo code for both of the functions is provided below:

Determine the **current state**
Check if there are **possible next states** for the **action**
If no **possible next states** exists:
- Return **current state, reward(0)** and **completion states(done)**

Select a **next state** randomly and mark it as active
Extract **selected cards**
Calculate the **reward** based on changes in selected cards between current and next states
If **action** is "pick":
- Increment the **card selection count**

If the **card selection count** reaches the limit:
- Mark the **completion status** as done
Return the **next state**, **calculated reward**, and **completion status**

Figure 6: Pseudo code of Step function

Reset the **current state**
Randomly select and mark a **state** as active
Extract **selected cards** and update the **selected cards counter**
Return the **selected state**

Figure 7: Pseudo code of Reset function

File implementing SARSA (SARSA.py)

This file, as its name implies, includes all the essential functions required for implementing the reinforcement learning algorithm SARSA, which stands for State-Action-Reward-State-Action, on a specifically defined custom environment. The initial step involves using the functions defined in the previous file to create an environment suitable for the algorithm's implementation. Once the environment is established, the agent is trained over a series of episodes within the environment. Each episode consists of individual steps designed to assist the agent in populating the Q table values, which identify the action that yields the best reward for a given state. Training the agent involves a sequence of steps, including selecting an action, taking a step based on the chosen action, observing the resulting state, and updating the Q table with the obtained values. A maximum number of steps is predefined, and the sequence of actions is repeated for this fixed number of steps or until a terminal state is reached, signifying that no further state can be achieved from the current state. Before the initiation of each training episode, the environment is reset and provided to the agent to enable a fresh start. The `'train'` function, defined in the file, encompasses all the logic necessary for training the agent within the defined environment.

Another function, `'choose_action'`, has been created to select an action based on an epsilon value, introducing randomness into the action selection process. This randomness ensures that all actions for all defined states are explored. Additionally, there is an `'update'` function within the file, responsible for modifying the Q-value in the Q table for

the current state-action pair. As every reinforcement learning algorithm relies on a specific formula to calculate the Q-values for the state-action pair, the update function contains the necessary code to incorporate this formula for calculating and updating the Q-values based on the provided inputs.

The SARSA algorithm is a reinforcement learning algorithm that aims to update Q-values in a way that balances immediate rewards with expected future rewards based on the agent's current policy. These Q-values represent the expected cumulative reward of taking a particular action in a particular state and are crucial for making decisions in reinforcement learning tasks. The update rule for SARSA is given by:

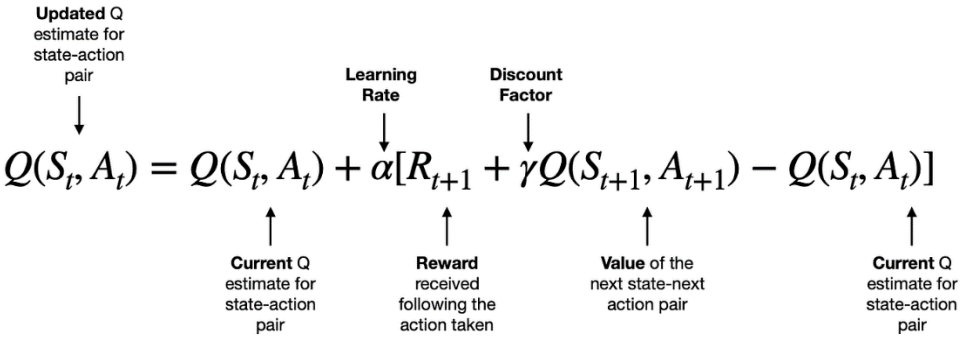

Figure 8: Update rule formula for SARSA algorithm

In this equation, Q (S_t, A_t) is the current Q-value for taking action A_t in state S_t, α is the learning rate, R_{t+1} is the immediate reward, and γ is the discount factor, which reflects the importance of future rewards. The term Q (S_{t+1}, A_{t+1}) represents the Q-value for the next state S_{t+1} and the action A_{t+1} that the agent actually takes. The value α, the learning rate, influences how much the Q-value is adjusted according to new information. A higher learning rate accelerates learning but can introduce instability, whereas a lower learning rate encourages more

reliable but slower learning. The discount factor, γ, dictates the significance of future rewards. A value near 0 prioritizes immediate rewards, whereas a value near 1 emphasizes long-term rewards more. The difference between the current Q-value and the target value (immediate reward plus discounted future reward) is known as the temporal difference (TD) error. This error drives the adjustment of the Q-value to improve the agent's policy over time. It's important to note that SARSA is an on-policy method, meaning that it updates Q-values based on the actions the agent actually takes while following its current policy.

Suppose the agent starts with the Q-value of 0 for all the state-action pairs. Assuming a learning rate of 0.1 and a discount factor of 0.9 are incorporated for the agent's training, Let the current state be denoted as (3|), meaning 3 being the current card with no cards selected for the current state. Let the reward be calculated by subtracting the selected cards from either of the states. DISCARD action leads us to a different state, say (1|) while PICK action leads us to (1|3) and the SWAP action leads us to the same state. The image below shows the calculation of Q-values, both initial and updated, using the update rule for SARSA algorithm.

Initial Q-value for all state-action pairs, thus Q(St, At) = 0 & Q(St+1, At+1) = 0
Learning rate, $\alpha = 0.1$
Discount rate, $\gamma = 0.9$
Reward = selected cards of state2 - selected cards of state 1

Assuming the next state is a PICK action, reward = (1|3) - (3|) = 3 - 0 = 3
Q(St, At) = 0 + 0.1[3 + 0.9*0 - 0] = 0.3
Q("(3|)", PICK) = 0.3

let the next state for the same PICK action be (2|3, 1),
reward = (2|3,1) - (1|3) = (3+1) - 3 = 1
Q(St, At) = 0.3 + 0.1[1 + 0.9*0 - 0.3] = 0.37
Q("(1|3)", PICK) = 0.37

Now, if the same state (3|) is obtained in a different episode of training, with PICK action,
Since we calculated that for Q("(3|)", PICK),
Q(St, At) = Q("(3|)", PICK)
Q("(3|)", PICK) = Q("(3|)", PICK) + 0.1[Reward + 0.9*Q("(1|3)", PICK) - Q("(3|)", PICK)]
Q("(3|)", PICK) = 0.3 + 0.1[1 + 0.9 * 0.37 - 0.3] = 0.4033

Figure 9: Example of update rule

Over the course of thousands of episodes, the Q-table, which stores the Q-values for each state-action pair, gets populated appropriately, with the best action for a given state having a high Q-value. This indicates the preference of that action over others in that state. The pseudo code for the functions `update`, `choose_action` and `train` are provided below:

Predicted Q-value = **current Q-value** for the given **state** and **action**.
Target Q-value = **reward** + Q-value for the **next state** and **action** * **gamma**
Final Q-value for the **current state** and **action** = **current Q-value** for the given **state** and **action** + **learning rate (alpha)** * (**target** - **prediction**)

Figure 10: Pseudo code for Updating the Q value in Q table

```
Generate a random number between 0 and 1
if generated random number () < epsilon:
  action = select a random action from the environment
else:
  action = get action of the max value in row denoted by (Q[state])
Return action
```

Figure 11: Pseudo code for choosing an action for given state

```
for episode from 0 to total episodes - 1:
  - Initialize state1
  - action1 = choose action based on state1

  for step from 0 to max steps - 1:
    - Use the Step function of environment to take an action
    - Get state2, reward and process status
    - action2 = choose action based on state2
    - Update Q value for (state1, action1) based on the values from (state2, action2)
    - state1 = state2
    - action1 = action2

    if process status = done:
      Terminate loop
Return Q Table
```

Figure 12: Pseudo code for training the agent

The file contains a number of additional functions, including those used to display the Q-matrix and Q-table as needed during the process. The final function in the file is the `calculate_reward` function, which is tasked with determining the reward for transitioning from one state to another. It takes two states, the current and next states, as input and uses the selected cards sections to calculate the reward value, which it then returns.

Main file (main.py)

The main file serves as the starting point of the project and is crucial for its execution. It is responsible for importing all the necessary files and utilizing the defined functions to train the agent. Once trained, the agent is then able to play the game. The key functionality implemented in this file revolves around the use of the Q-table and the agent to facilitate gameplay. To initiate the game, the following steps are taken:

1. Calculating the maximum reward using the cards in the deck.
2. Shuffling the deck of cards.
3. Drawing a card from the shuffled deck.
4. Referring to the populated Q-table to obtain the action for all the states following the current state, in order to reach the next state and simulate gameplay.

The termination condition occurs when the selected cards achieve the maximum reward calculated earlier. After obtaining the first state from the shuffled deck, the action with the maximum value from the Q-table is determined for that state. Subsequently, a state is randomly selected from the list of states within the dictionary of possible states, which was previously calculated in the *custom_environment.py* file, to obtain the next state. A reward value is calculated based on the transition from the current state to the next state. If this reward value is negative or equal to zero, another state from the list of states is selected. Otherwise, the current state is chosen as the next state. If there are no states left in the list of possible states, the next best action for the current state is chosen and its possible states are explored. This process ensures that the agent reaches a state with the maximum reward

value, as no other state seems to be rewarding regardless of the action taken by the agent.

The pseudo code for this `play` function is shown below:

```
Initialize max_reward
Randomize the order of cards in the deck (shuffle)
Initialize selected_cards, card_in_hand, and current_state
Set current_state by picking up a card from deck

while condition is TRUE:
    - Get Q_values for current_state from Q table
    - Sort these Q_values and reverse them

    for each action with high Q_value:
        - Get possible states for action

        if possible states are not empty:
            for each of these possible state:
                - Calculate reward for the state
                if reward is positive:
                    - Update reward and append this state in selected_cards
                    - Update current_state
                    - Terminate the loop

        if  no state is selected:
            - Skip the loop for current action and go to next action

    Update selected_cards from current_state
    if selected_cards meet criteria & reward matches max_reward:
        - Set condition as FALSE
```

Figure 13: Pseudo code for Play functionality

Conclusions and Future work

In conclusion, implementing this project laid a solid foundation for understanding reinforcement learning (RL) and its diverse algorithms, demonstrating how different methods can be tailored to suit the specific needs of AI in various scenarios. The challenges faced during the implementation have provided a better understanding of how different approaches to a particular problem can provide conflicting and complex solutions, highlighting the importance of choosing the correct approach and algorithms. Through hands-on experience, the project illustrated how RL can be implemented across a broad spectrum of applications, from simple card games to sophisticated enemy AI in cross-platform games, underscoring the versatility and power of reinforcement learning in real-world problem-solving. In addition, the project requires sufficient refactoring and re-writing for improved efficiency, with a focus on minimizing the use of multiple loops as well as targeting complex problems or game types. In addition to restructuring the code, there is significant potential for enhancing the project by changing the representation of necessary variables and values, such as the states or the state-action pairs, thus achieving optimal performance by the algorithms.

References

- Kaelbling, L.P., Littman, M.L., Moore, A.W. (1995). “An Introduction to Reinforcement Learning”, Journal of Artificial Intelligence Research. Retrieved from https://link.springer.com/chapter/10.1007/978-3-642-79629-6_5
- M. Naeem, S. T. H. Rizvi and A. Coronato, "A Gentle Introduction to Reinforcement Learning and its Application in Different Fields,". *IEEE Access*, vol. 8. Retrieved from https://ieeexplore.ieee.org/abstract/document/9261348
- R. S. Sutton and A. G. Barto. (1998). “Reinforcement Learning: An Introduction”. MIT Press, 1998. Retrieved from https://mitpress.mit.edu/9780262039246/reinforcement-learning/