Final Project

An Artificial Intelligence Model to Play UNO, a Turn-Based Card Game

AE4350: Bio-Inspired Intelligence for Aerospace Applications

Daniel Mena

Delft University of Technology

TU Delft

Final Project

An Artificial Intelligence Model to Play UNO, a Turn-Based Card Game

by

Daniel Mena

Student Name	Student Number
Daniel Mena	5900980

Instructor:	Dr. G.C.H.E. de Croon
Instructor:	Dr.ir. E. van Kampen
Project Duration:	February, 2023 - August, 2023
Faculty:	Faculty of Aerospace Engineering, Delft

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Contents

1

Introduction

UNO is a common turn-based party game that originated in 1971 by an American barber named Merle Robbins [6]. Robbins created the game due to a dispute about the rules with his son over a similar game called Crazy Eights. Since then, UNO has become one of the most popular card games in the world, selling millions of units every year[5].

UNO is a fairly simple game, with the primary goal being for a player to use all of their cards before their opponents[6]. Certain cards are compatible with others, and special cards can be placed to make the game more interesting.

Due to the game's simplicity, the game can be tackled with a simple autonomous script that decides the cards that are compatible in a certain turn and selects a random one to place on the deck [1]. However, these plays can be optimized with artificial intelligence to increase the winning rate of the player. This is the main objective of this project.

The GitHub repository for the project can be accessed via the following URL: `https://github.com/dandemo212/UNO`

2

Rules of the Game

The main objective of UNO is for a player to use all of their cards before their opponents[6]. Every player is dealt seven cards at the beginning of each game, and a player progresses by placing a compatible card in the discard pile during their turn. In the case that they do not have any compatible cards, they must draw a card from the draw pile.

If the drawn card is compatible, the player is able to place it directly in the discard pile and end their turn. Otherwise, the drawn card is added to the player's deck and their turn ends.

An UNO deck consists of 108 cards[7]:

- 4 wild cards
- 4 wild +4 cards
- 76 number cards
- 8 skip cards
- 8 reverse cards
- 8 draw 2 cards

2.1. Wild Cards

Wild cards are able to be placed after any card. These cards change the color on the board to one of the user's discretion.

2.2. Wild +4 Cards

Similar to wild cards, these cards can be placed after any card. Playing this card results in the next player having to draw four cards from the discard pile. The next player's turn is also skipped. Due to the hardware limitations of the machine being used for this project, this card has been omitted from the AI model's training process.

2.3. Number Cards

An UNO deck consists of 76 number cards, 19 of each color: red, blue, green, and yellow [7]. For each color, there is one card of number 0 and two cards of each number 1-9.

A player can use a number card on top of any card that has the same color. The number card can also be used on top of any card with the same number, regardless of the color.

2.4. Skip Cards

An UNO deck consists of eight skip cards, two of each color. The skip card skips the turn of the next player. These cards are colored, so they can only be placed on top of other cards of the same color, or other skip cards.

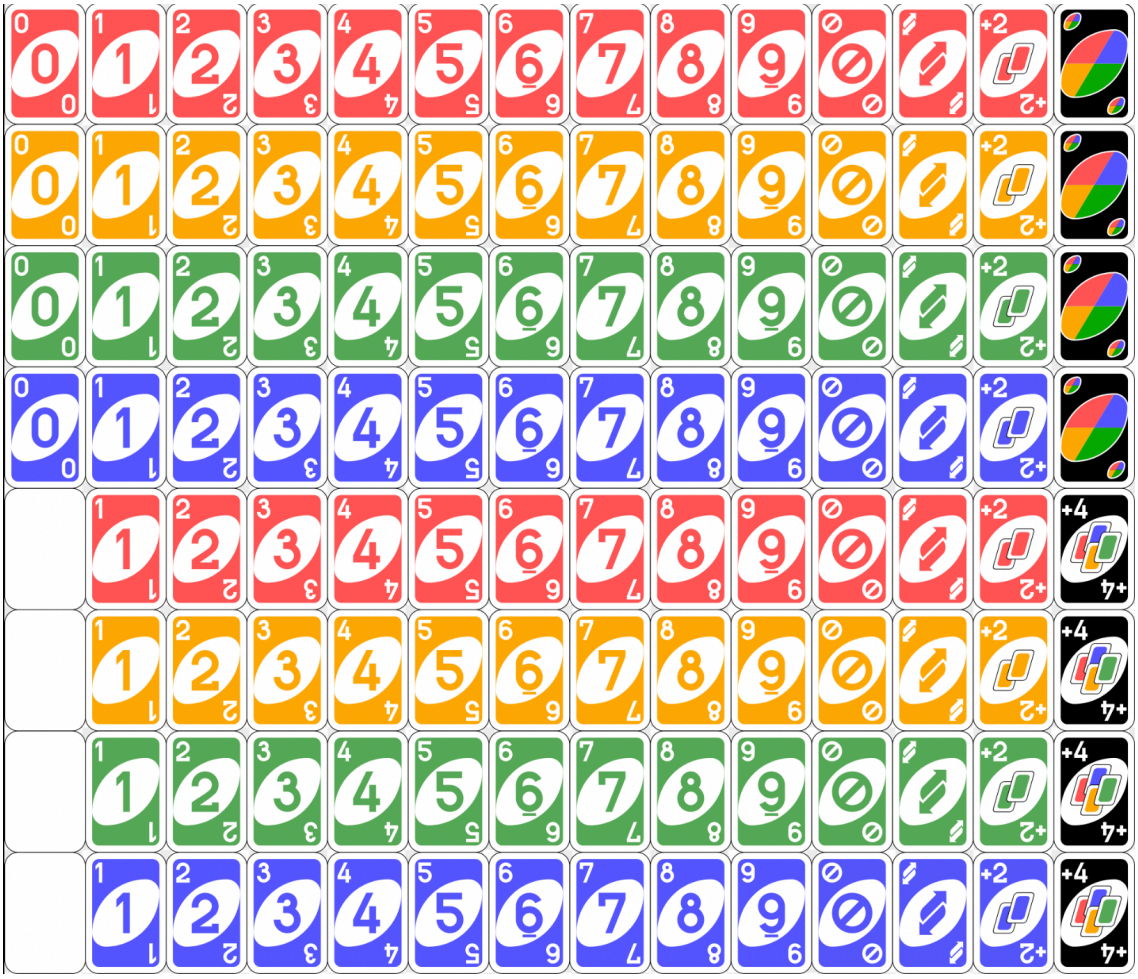

Figure 2.1: A standard UNO deck consists of 108 cards. Photo: Wikimedia Commons

2.5. Reverse Cards

An UNO deck includes eight reverse cards, two of each color. Reverse cards change the flow direction of the game. For example, if the ordering of the turn is clockwise, the usage of this card would shift the direction to be counter-clockwise. Using this card constitutes the end of the player's turn. This card can be placed on top of any other reverse card or any card of the same color.

2.6. Colored +2 Cards

The deck contains eight +2 cards, two of each color. These cards operate similarly to Wild +4 cards, however their usage rules are the same as the Reverse, Skip, and Numbered cards. These +2 cards can only be used on top of another +2 card or a card of the same color. Due to hardware limitations, these cards have been omitted from the deck.

2.7. Stacking

Colored +2 Cards and Wild +4 cards can be stacked. For example, if a player places a +2 card, the next player can place another +2 card and not have to draw.

3

The Agent

The model and all of its components are built in Python. The model utilizes the following classes for its foundational game functionality, then derives its decision-making inputs from the current state of the board.

- **Game** - Constitutes the overarching framework for a game. Creates players, builds the deck, shuffles the cards, and distributes them accordingly. Manages the discord pile and the reshuffling of cards when the draw pile becomes empty.
- **Card** - The most primitive class type. This class contains information about a card's type, color, and number value, if applicable. These properties are stored in Python enumerators *CardColor* and *CardType*. The class contains functionality to check the properties, change the properties (useful for wild cards), and determine compatibility with another card.
- **Deck** - This class is comprised of a Python list of *Card*s. The deck contains functionality for shuffling, drawing, and peeking. A peek allows a card at a certain index to be viewed without being removed, while a draw returns the card from the deck.
- **Action** - This class concerns the bridge between the game and the agent. Through this class, certain inputs are used to determine the policy, and the rewards are returned to the agent for decision-making.

3.0.1. Game

Method	Inputs	Functionality
deck_sizes		Prints the sizes of every player's deck to the console.
all_decks		Returns the *deck* local variable.
get_draw_history		Returns a list containing the draw history for each player.
get_top_card		*peek*s at the top card in the deck.
get_direction		Returns the local variable corresponding to the direction of play of the UNO game.
ai_deck		Returns the deck of the agent.
get_draw_pile		Returns the draw pile.
get_discard_pile		Returns the discard pile.
get_compatible_cards	my_cards: Deck	Returns a list of cards in the player's deck that is compatible with the last card played to the discard pile.
play_turn	idx: int	Plays the card of the agent corresponding to the given index *idx* and plays the cards of all other players randomly.

3.0.2. Card

Method	Inputs	Functionality
INIT		Prints the sizes of every player's deck to the console.
get_color		Returns the color of the card.
set_color	color: CardColor	Sets the color of the card. This method is usually used for wild cards.
get_type		Returns the type of the card.
get_number		Returns the number on the card. -1 is returned if this field is not applicable.

3.0.3. Deck

Method	Inputs	Functionality
get_cards		Returns a list of the cards in a deck.
add_card	card: Card	Adds a card to the deck.
shuffle		Shuffles the deck using NumPy's *np.random.shuffle()* method.
draw		Removes a card from the top of the deck and returns it for reference.
draw_at	idx: int	Removes a card from the given index *idx* in the deck and returns it for reference.
peek		Returns the top card from the deck without removing it.
peek_at	idx:int	Returns the card at the given index *idx* without removing it.
get_size		Returns the size of the deck.
build_deck	shuffled:bool	Creates a new deck based on the requirements defined in Chapter 2. Shuffles the deck according to the given argument *shuffled*.

3.0.4. Inputs

The agent utilizes an Epsilon-Greedy approach with Q-Learning strategy to determine the best-case scenario from a list of possible cards. Given an *Action* object with data about the number of cards each player has, draw history, direction of play, and playable cards, the policy function is able to determine the rewards of each card.

The policy sets every card's reward to -2 by default, then modifies each card's reward individually depending on a number of inputs. The following inputs are considered when determining the rewards for using each card:

- **How many cards the next player has** - If the next player has fewer cards than the agent, the policy will favor a card that will negatively affect their hand, such as a skip, reverse, or draw card.
- **What card the next player recently drew** - Players only draw cards when they do not have any compatible cards. The AI recalls what the last drawn card from the next player is so that, in the case that it decides to play aggressively, it can change color to whatever the next player does not have.
- **How many cards the previous player has** - If the previous player has fewer cards than the agent, the policy function will not favor a reverse card, as this will enable them to deplete their cards sooner, ultimately decreasing the likelihood of the agent winning.
- **What cards the agent is able to play** - If the agent cannot play a card, it will not consider it when deciding what card to play. The agent will keep the card's reward at -2 and disregard it for the turn.
- **Unplayable cards in the agent's deck** - Although only the compatible cards are able to be played in a given turn, the agent still takes into account the other cards in its deck. If a majority of the agent's cards are a certain color, the policy will favor another card that is able to change the color to the color that most of the agent's hand is in.

Within each input group listed above, four to five inputs from the game are used to determine the current state of the board and make proper decisions [2]. The rewards were calculated with each of these inputs, with a positive influence resulting in an increase of one point to the card in the deck and a negative influence resulting in a decrease of one point.

3.1. Wild Cards and Wild +4 Cards

Wild cards will always be favored to be played last. This is because if a player's last card is a wild card, the probability that it will be compatible with the card played by the previous player is one hundred percent. However, if there are no compatible cards, the policy will prefer that the agent use the wild card rather than drawing a new card.

3.2. The Epsilon-Greedy Approach

The Epsilon-Greedy approach refers to an algorithm that exploits the highest reward possibility of a known approach with a probability ϵ and explores a random approach with probability $1 - \epsilon$ [4]. In order to ensure a sufficient exploration-exploitation ratio, an exploration decay factor ε must be defined. After each iteration, ϵ is decreased to encourage the agent to exploit the information it already knows to find better solutions [4].

Given a minimum epsilon value ϵ_{min}, maximum epsilon value ϵ_{max}, and episode timestep *x*, the equation representing the change in epsilon after each iteration can be defined as:

$$\epsilon = \epsilon_{min} + \epsilon_{max} - (\epsilon_{min} * e^{-\varepsilon x}) \tag{3.1}$$

Using this approach, varied ε were tested to give the agent the best chance of winning the game [4].

3.3. The Q-Table

A Q-Table refers to the data structure used to store and recall weighted values corresponding to the state-action pair considerations made by the agent [3].

The agent uses a Python table data structure to store the values for the Q-Table. On the vertical axis, the agent stores information about the state: the card that is currently on the board. On the horizontal axis, the agent stores information about the action: what card can be placed in response. There are 51 columns and 51 rows in the Q-Table, each representing the possible states and actions, respectively.

Because cards cannot be used as indices for Python tables, a function is used to convert each card type to a unique identifier that can then be compared with other cards to determine their equivalence. These identifiers are created as an MD5 string comprised of the following properties:

For **Wild Cards** and **Wild +4 Cards**, the string identifier is composed of the *CardType* enumerator of the card. For **the action of drawing a card**, the identifier is simply the string *DRAW* hashed in MD5. For **all other cards**, the string is a result of the concatenation between the card type, card color, and card number.

When updating a Q-Table, it is important that the new information is stored in such a way that it preserves the previously determined data [3]. This ratio that describes the influence of new data on the agent's memory is called the **Learning Rate** and is defined by α. Another constant, the discount rate γ represents the agent's propensity to take future consequences into account. This value γ functions to diminish the reward over time. The learning rate α and discount factor γ can be related to the values of the Q-table *q* with state *s* and action *a* in the following manner:

$$q(s', a') = (1 - \alpha) * q(s, a) + \alpha * (R_{s,a} + \gamma * \max[\sum R_{s'}]) \tag{3.2}$$

where *s'* and *a'* represent the new state and action pair values. This equation can also be referred to as the Bellman Equation [3].

3.4. Recognizing an Alternative Approach

An alternative approach also has the potential to produce effective results. This approach was shared in "Winning Uno With Reinforcement Learning" by O. Brown, D. Jasson, and A. Swarnakar, where the policy was designed from an omniscient standpoint. Rather than assessing the cards and providing rewards in a turn-based frequency, their approach provides rewards depending on the outcome of the game: +1 for winning, -1 for losing, and 0 for all intermediate states [2].

Although this approach is functional, it would require many more sample games, which is computationally expensive. However, this approach provides more freedom to the agent, allowing it to reach an optimal win percentage with fewer trials. Instead of only using the cards recommended by the policy,

the agent is able to explore and exploit at its own discretion, presenting the possibility for faster learning overall.

4

Results

Due to the generally random structure of UNO, the agent's results depended largely on the number of cards at play [1]. However, over time, the agent considerably increased its average reward.

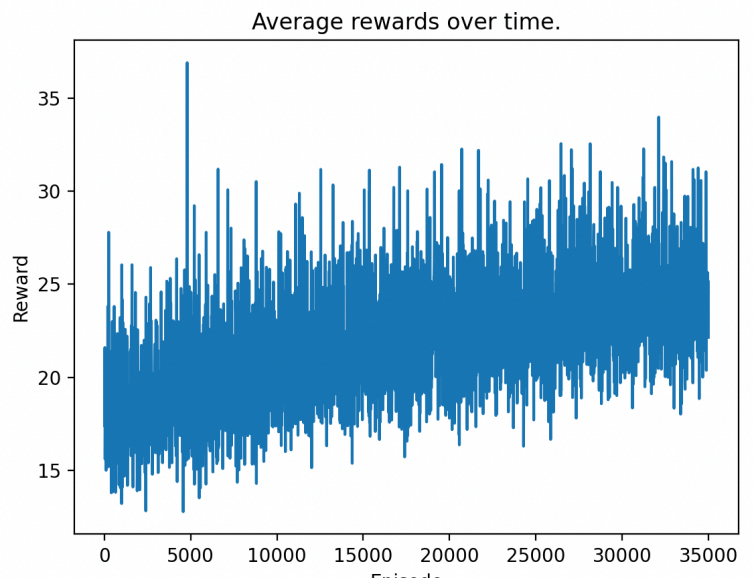

Figure 4.1: Average reward over time in 10-episode increments for 35000 episodes.

The average reward increased from 18.872 to 26.272, as seen in Figure 4.1. The outlier at the 5000-episode mark is likely due to a lucky setup of the UNO game, as numerous re-runs of the script result in no outlier being displayed.

The agent's wins also increased as it began learning more information, as it was able to better decide how to perceive certain states. This can be seen in Figure 4.2.

4.1. Uncertainty

Multiple runs were performed in order to determine the stability of the results and detect potential sources of uncertainty. Of five runs with 10,000 episodes each, the average improvement in rewards score is shown in Table 4.1.

During every run, there was a considerable improvement in the agent's average total reward. This is due to the agent's increased tendency to exploit rather than explore over time.

Thus, the general uncertainty has been addressed, as the results consistently show the influence of the agent's Q-learning process on the outcome of the UNO game.

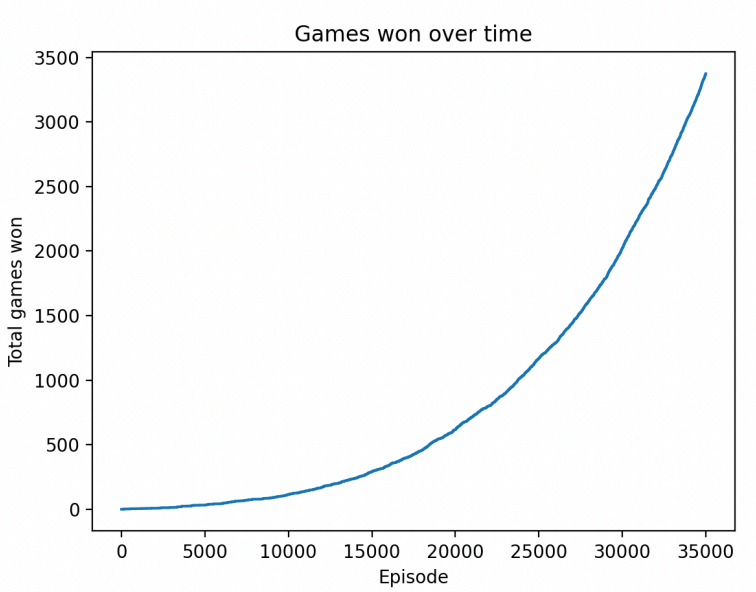

Figure 4.2: Games won over time.

Trial	Initial R_{avg}	Final R_{avg}
CONTROL	18.872	26.272
1	13.500	23.192
2	14.500	24.378
3	18.000	26.118
4	18.500	26.801
5	16.600	22.547

Table 4.1: Average improvement of average reward R_avg over five trial episodes.

4.2. Sensitivity Analysis

It is also important to determine the stability of certain algorithms in order to quantify the effect that each variable has on the overall solution. By modifying the learning rate α and discount factor γ with 5000-episode samples, the effect of each variable can be determined independently.

As can be seen in Figure 4.3, the final reward increases with a generally linear trend, less occasional deviations from the line of best fit between the data points. This deviation is caused due to the randomized nature of UNO, suggesting that a more skill-based game will provide a conventionally linear curve. The variable is therefore stable, as slight perturbations to the data will only result in slight deviations from the target value.

Although the graph in Figure 4.4 can also be related to a positive linear regression, it is much less stable. The graph oscillates greatly about the line of best fit, making predictions based on data points much less accurate. This is likely due to the luck-based components of the UNO game coupled with the large influence the discount factor has over the behavior of the agent, as described by equation 3.1.

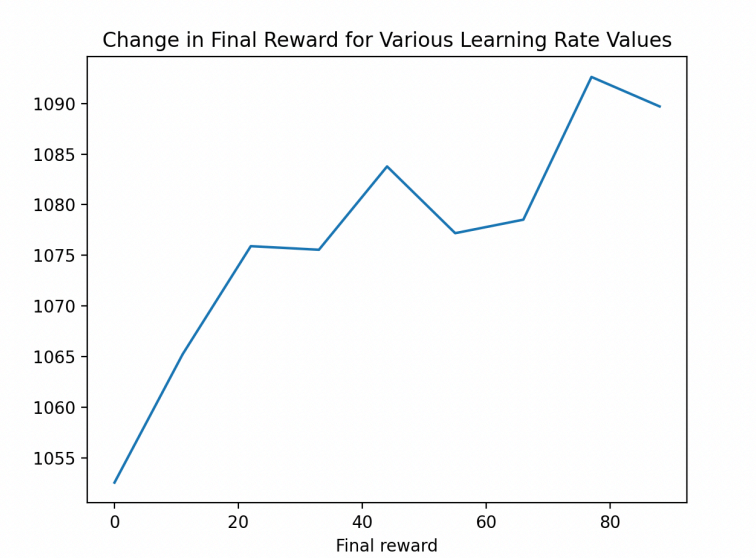

Figure 4.3: Effect of varied learn rate α on overall agent.

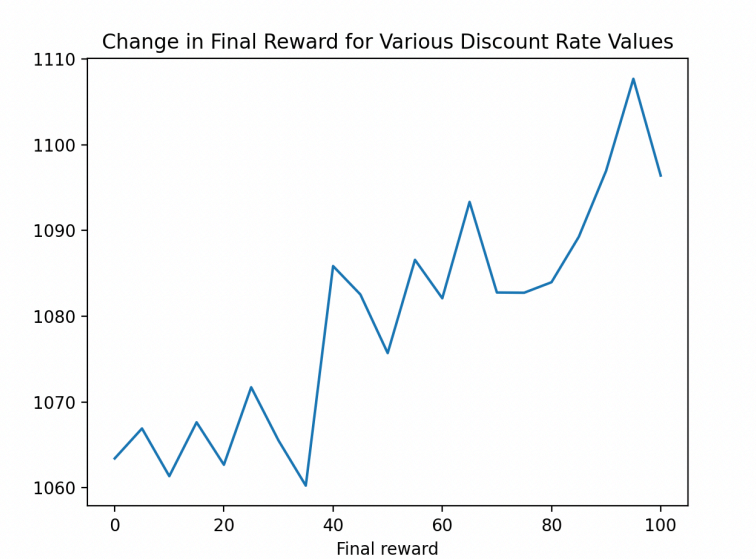

Figure 4.4: Effect of varied discount factor γ on overall agent reward.

References

[1] N.U. Maulidevi A. Ramadhan H. Iida. “Game Refinement Theory and Multiplayer Games: Case Study Using UNO”. In: (2015). URL: https://dspace.jaist.ac.jp/dspace/handle/10119/12881.

[2] O. Brown, D. Jasson, and A. Swarnakar. “Winning Uno With Reinforcement Learning”. In: (2020).

[3] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (Mar. 2016). DOI: 10.1609/aaai.v30i1.10295. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10295.

[4] M. Babes M. Wunder M. Littman. “Classes of Multiagent Q-learning Dynamics with E-greedy Exploration”. In: (2010). URL: http://engr.case.edu/ray_soumya/mlrg/2010%20Classes%20of%20Multiagent%20Q-learning%20Dynamics%20with%20e-greedy%20Exploration.pdf.

[5] The Strong National Museum of Play. *America’s Favorite Game and Success Story: Uno!* URL: https://www.museumofplay.org/blog/americas-favorite-game-and-success-story-uno/.

[6] UNO Rules. *UNO History*. URL: https://www.unorules.org/uno-history/.

[7] Let’s Play UNO. *UNO Cards*. 2023. URL: https://www.letsplayuno.com/news/guide/20181213/30092_732567.html.