Final Project: UNO Bot

By: Luciana Diaz-Albadan & Geneva Ng

Project Overview:

For our Visual Interfaces final project, we have created an assisted UNO gameplay system that we have called UNO Bot. The purpose of the UNO Bot is to suggest to its user possible gameplays during a given round. Capable of recognizing most[1] cards within a traditional 108-card UNO deck, our UNO Bot takes two inputs of live video feed: the live deck and the current hand. Deploying an object detection model upon the visual input, the bot recognizes all presented cards and returns a legal UNO move in the form of a string. Examples of possible outcomes include:

hand [['green', None, 'draw2'], ['green', None, 'draw2'], ['green', '5', None]]
deck ['red', '3', None]
draw another card!

hand [['yellow', None, 'draw4'], ['blue', None, 'reverse'], ['blue', None, 'draw4']]
deck ['red', None, 'skip']
sorry, turn skipped

hand [['wild', None, None], ['green', None, 'draw4'], ['green', '1', None]]
deck ['green', '1', None]
put down this card: ['wild', None, None]

hand [['red', None, 'draw4'], ['green', '3', None], ['green', '8', None]]
deck ['green', '1', None]
put down this card: ['green', '3', None]

Various manners of implementation and domain engineering were tested to determine the best way to seamlessly integrate the functionality of the UNO bot during a traditional[2] gameplay setting. The following report will delve into the research, testing, and observations gathered from the project development process, which has been broken down into three phases: object detection model development, model deployment, and game logic implementation. Considering all members of the project had little to no previous experience

[1] The limitations of card input will be discussed further in subsequent sections.
[2] “Traditional” refers to the original UNO gameplay

dealing with live object detection and machine training, each phase has in itself a trial-and-error learning process that ultimately serves as an explanation for our final design implementations.

Object Detection Dataset Development

For the primary visual interface aspect of our project, we chose to design an object detection model that would take a visual form of input to detect and classify any of the 54 unique cards in an UNO deck. During the model development process, various different approaches regarding annotation technique, dataset size, class expansion, dataset preprocessing, and model types were experimented with to determine the best fit for our project.

For the annotation process, we utilized Roboflow, a cloud-based platform that provides a suite of tools and services for computer vision developers. Its main functionality is to help users create and manage datasets for training machine learning models for image and video recognition tasks. The Roboflow dataset generation process can be summarized as follows:

1. Upload images
2. *For each image*: annotate based on the bounding boxes desired for the object detection model to be trained upon, making sure to assign each box to its designated class.
3. Select a training, validation, and test split (%) of the dataset
4. Expand the dataset by applying different preprocessing and augmentation selections
5. Export the dataset for model training

This overall dataset generation process was largely kept constant throughout all the different variations of models that were tested. What mainly changed throughout the varying iterations were details such as the bounding box area of interest, dataset size, preprocessing selections, and model type.

Trial 1: Whole Card Recognition

Our first model utilized bounding boxes that recognized the entire uno card. In this trial, the following 7 class labels were generated within our dataset:

green_card
yellow_card
blue_card
red_card
wild_card
draw_four

uno_card

All images were generated by Luciana, taken in her dorm. Approximately 250 pictures were uploaded into the Roboflow system for manual annotation. Examples of pictures found within this dataset are as follows:



These images were annotated as such:



The train/valid split generated 177 training images (75%), 27 valid images (~11%), and 32 test images (14%). This train/split percentage designation was recommended by the Roboflow platform. As first time users, we decided to go with this suggestion.

Following the annotation process, the training image dataset was expanded using the following augmentation steps (provided in a dropdown selection by the Roboflow platform):

- 90 degree rotation (clockwise, counterclockwise, upside down)
- 30 degree (clockwise, counterclockwise, upside down)
- Shear (45 degree horizontal & 45 degree vertical)
- Saturation
- Brightness
- Exposure
- Bounding box brightness

- Bounding box blur

Two major flaws in the creation of this database were recognized prior to model training implementation. The first was that the cards were classified only by color. In designing such a limited class set, it was obvious that a successful game of UNO was not going to be able to be played as card numbers and actions were not going to be detected. The second limitation was that there was little to no variation in the background on which the cards were staged, meaning there were unlikely chances for the deployed model to work successfully on a background that differed from that provided in the dataset. This was an unintentional oversight as this first trial was ultimately a learning experience to better understand the necessities of a comprehensive dataset.

To better understand how successful object detection databases were generated, we turned to Roboflow's Universe feature. Students can learn from Roboflow Universe to create better object detection datasets by studying the structure of the dataset and the types of images included. By studying the diversity of images in the dataset, they can also learn how to build datasets that are more representative of real-world scenarios. Additionally, they can use the pre-built data processing workflows in Roboflow to clean and augment datasets, which can improve the accuracy of generated models.

After studying various datasets, such as this one (https://public.roboflow.com/object-detection/uno-cards) suggested by the TA in our project proposal feedback, we decided to scrap this dataset and implement a new approach.

Trial 2: Corner Card Recognition

Clearly, our previous attempt of recognizing the entire card was not a productive use of the model as it would not pick up on different card numbers and actions. At this stage, we were faced with the following question: how can we recognize a card's number *and* color? Studying previous databases, especially the one mentioned earlier, we noticed that the images had an extremely wide range of backgrounds on which the cards were juxtaposed on. Moreover, the cards were positioned in a three-card fan, which (at first glance) seemed to have the possibility of successful integration with our proposed implementation, since we planned on having the cards displayed as a fan as well.

Taking these observations into consideration, we decided to take an annotation approach that concentrated on the top left and bottom right corners of the UNO card. Considering we wanted to detect both number and color, the following 54 classes were selected:

green_0	blue_5	red_rev
green_1	blue_6	red_draw
green_2	blue_7	red_skip
green_3	blue_8	yellow_0
green_4	blue_9	yellow_1
green_5	blue_rev	yellow_2
green_6	blue_draw	yellow_3
green_7	blue_skip	yellow_4
green_8	red_0	yellow_5
green_9	red_1	yellow_6
green_rev	red_2	yellow_7
green_draw	red_3	yellow_8
green_skip	red_4	yellow_9
blue_0	red_5	yellow_rev
blue_1	red_6	yellow_draw
blue_2	red_7	yellow_skip
blue_3	red_8	wild_card
blue_4	red_9	draw_four

Considering 54 classes is quite a lot of different classifications to identify, we realized we would need a much larger dataset compared to our previous file of 250 images. To make best use of our time, we decided to download the raw image files of the public[3] Uno Cards database and annotate them according to our own needs. Examples[4] include:

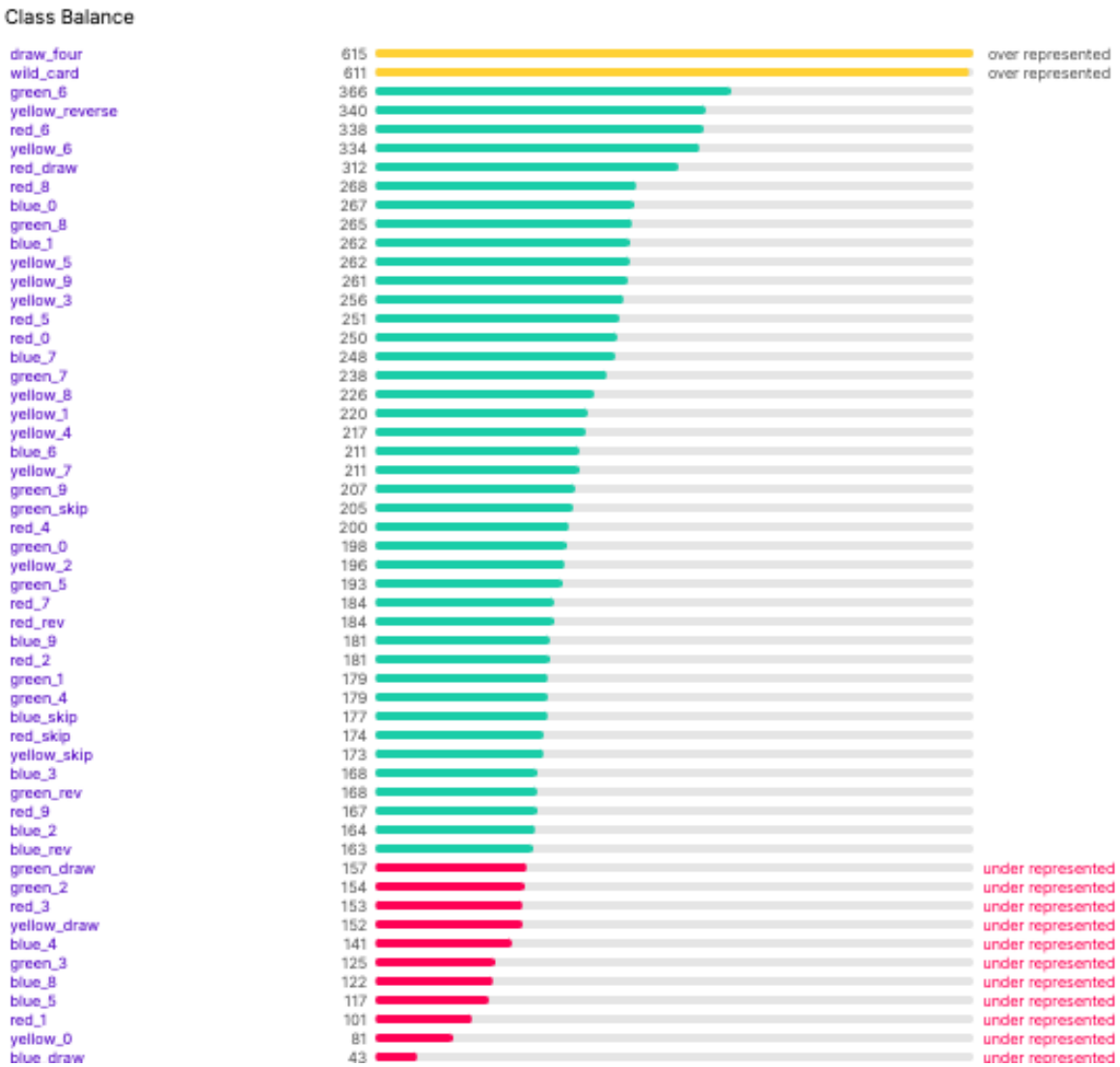


[3] Let it be noted that the incorporation of a public data set within our project was an idea suggested to us by the TA who graded our project proposal. The dataset we utilized has been authorized for public use, as stated in its documentation as well as Roboflow's open source guidelines.

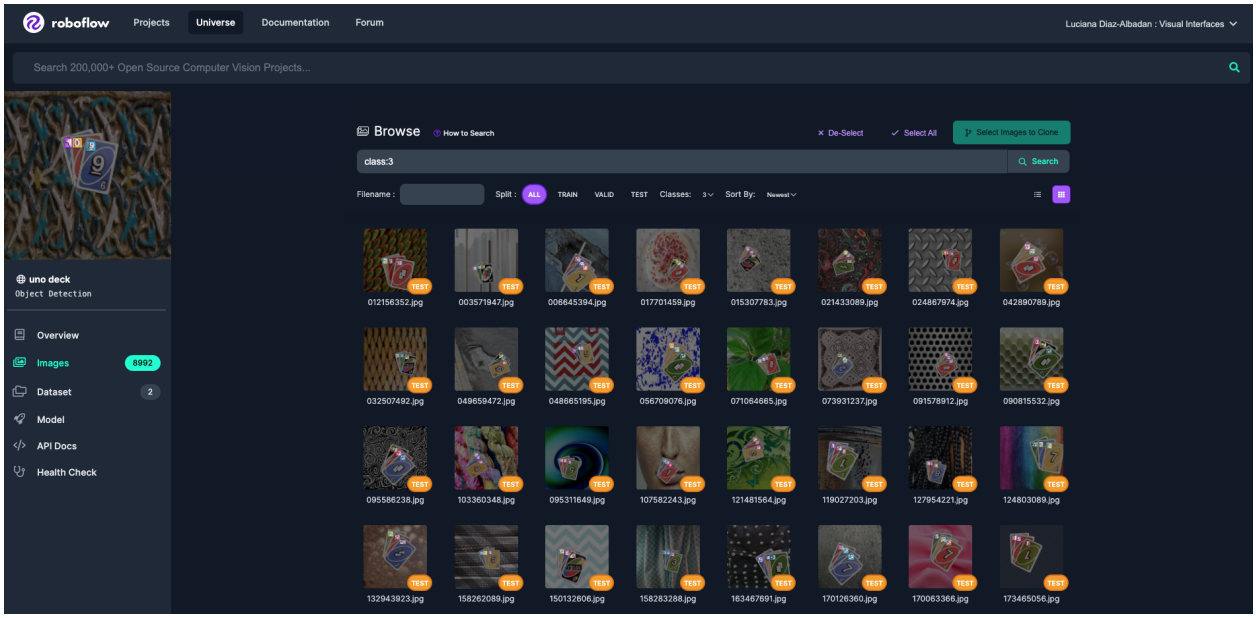
[4] The following images appear darker than what they actually are. The Roboflow annotation platform darkens the images when screenshots are taken.

The images were annotated and classified based on their card and color/action. Top cards — meaning the first card in a fan, were given bounding boxes that covered all three numbers/actions displayed. This was done with the intention of increasing the amount of bounding boxes/class instances within the dataset and hopefully creating an even more accurate model. After all, when playing UNO, it does not matter the number of cards of a certain type that one has in one's hand since only one card can be played at any given time. The images in the dataset were annotated as such:

The same test/train percentage split that was utilized in the previous model was implemented on this dataset. As all the images in the public set seemed to have a randomized mix of all 54 unique UNO cards, Roboflow's Data Health Check feature was constantly monitored in order to make sense of how many card classifications for each class was being documented. If any class was flagged to be underrepresented, we would utilize Roboflow Universe's search filters to filter through the public dataset by number (as that's how the classes were designated for that particular project). After having the platform display the images by only number, we would comb through and select the necessary images that matched our needs (color-wise) and import the raw images into our project folder. The following two screenshots show (1) the health set that was continuously consulted to guide our image selection process and (2) an example of how the public dataset was filtered for specific selections.



Health Check Chart



Example: Filtering through the public data set to only show images that contained instances of a draw_2 card

The images above show an example of how we attempted to create a balanced database. In a case such as this one, where the blue_draw2 card is flagged as severely underrepresented, we would comb through the draw_2 filter in the public database, and pick the blue instances that are of relevance to our needs.

It should be noted that the annotation process is quite a time consuming process. In the case that we did not have time restrictions to adhere to, we would have created a much larger dataset than the finalized selection we settled upon. In the end, we generated 2512 manually annotated images throughout the various stages of the model development process.

At this point in the project, Luciana had been granted an academic research license by the Roboflow team (had applied weeks earlier). In having this access, we were able to augment our training dataset by x5 compared to the x3 amplification limitation that was allowed previously. The augmentations applied to this dataset were as follows:

AUGMENTATIONS	
	Outputs per training example: 5
	Flip: Horizontal, Vertical
	90° Rotate: Clockwise, Counter-Clockwise, Upside Down
	Rotation: Between -39° and +39°
	Shear: ±44° Horizontal, ±41° Vertical
	Exposure: Between -44% and +44%
	Blur: Up to 1.75px
	Noise: Up to 5% of pixels
	Cutout: 6 boxes with 11% size each
	Mosaic: Applied
	Bounding Box: Shear: ±45° Horizontal, ±20° Vertical

Augmentations Summary

These were all the *relevant* augmentations that were offered as options by the Roboflow system. Again, as mentioned previously, the maximum number of augmentations that were possible were selected in hopes of creating a diverse dataset and, in turn, a more efficient detection model.

Our final dataset included 8.8k test images, 482 validation images, and 266 test images. This dataset was exported and fed into a **YOLOv8** model for training.

Model Training

We selected the YOLOv8 object detection algorithm for our project because it is one of the most accurate and efficient object detection models available. We needed an algorithm that could accurately detect and classify objects in real-time, and YOLOv8's ability to process video frames at a high speed made it an ideal choice. Furthermore, YOLOv8's ability to handle small objects and detect multiple objects in a single frame was crucial to our project, as we needed to accurately detect and count small objects in real-time.

Despite the benefits of the YOLOv8 object detection algorithm, we faced significant challenges due to its slow training time. It took anywhere from 2 to 5 hours to train a single version of the model, which significantly impacted our ability to fine-tune the model with as much frequency as we would have liked. This in turn negatively impacted the model's performance, as we were not able to experiment with different hyperparameters and training configurations as much as we would have liked. In addition, the slow training time made it difficult to scale our project and train the model on larger datasets or more complex object detection tasks, such as detecting the cards in poor lighting. As a result, we had to carefully

balance the trade-off between training time and model performance throughout the development process.

We utilized Google Colab Notebook to train our YOLOv8 model. We selected Google Colab because it provides free access to powerful GPUs and a collaborative environment, making it an excellent choice for first-time object detection and machine learning students like ourselves. Additionally, Colab comes pre-installed with popular machine learning libraries like TensorFlow and PyTorch, eliminating the need for setting up a development environment. However, a huge drawback from using Google Colab is that the page would fail to save any progress made on training the YOLOv8 weights if the runtime got disconnected at any point during the train portion. This was a particularly relevant issue for us, since Google Colab typically disconnects after 90 minutes of idle activity. As we mentioned earlier, our models required much more than 90 minutes of training time. This posed a huge hurdle for our project, since it seemed like we could never run training versions with a substantial amount of epochs. It always seemed that whenever we tried to train our system for more than 20 epochs, the system would disconnect and all our progress would be lost. For this reason, any time that we wanted to train our system, we had to physically be by our computers for the entire duration of the training, making sure the system would not disconnect. This was extremely frustrating and inefficient, and there were multiple times that the system would unexpectedly fail around the 95% completion stage, despite our undivided attention. Ultimately, the maximum number of epochs we were able to train our model on was 40.

During our training portion, we generated nine different versions of our Object Detection model, each of them increasing in its mAP[5]. mAP, or mean Average Precision, was used as a "north star" guiding metric as it is commonly used to evaluate the performance of object detection models, measuring the accuracy of the model in terms of both precision and recall. Providing a single numerical score that summarizes the model's overall performance, we utilized this number to gauge how effective our dataset manipulation was being translated into model efficiency. In sum, the generation of these 9 versions totaled more than 25 hours of runtime.

VERSION	**mAP (%)**	**IMPROVEMENT FROM LAST VERSION**
1	43.6%	**BASE VERSION**

[5] The mAP metric is particularly relevant in object detection because it takes into account the different levels of confidence of the model in predicting the presence of objects in an image, and it penalizes incorrect predictions and false positives. A high mAP score indicates that the model can accurately detect and locate objects in images, while a low mAP score indicates that the model has a high rate of false positives or misses.

2	44.5%	Increased general dataset size
3	64.8%	Increased general dataset size
4	82.8%	Increased dataset of underrepresented classes
5	86.8%	Remapped card_9 classes
6	87.8%	Increased dataset of underrepresented classes
7	88.9%	Increased dataset of underrepresented classes
8	90.3%	Increased dataset of underrepresented classes
9	92.6%	Increased dataset of underrepresented classes

Versions 1-3 include datasets that were 600, 1200, and 1800 images in size respectively. These images were arbitrarily chosen and the data health check chart was not consulted. At this point, we were trying to familiarize ourselves with the YOLOv8 process, as well as the usage of the Google Colab platform. Version 4 was the first version in which the underrepresented class images selection process (as mentioned in the Dataset Development section) was implemented and trained. This helped increase the mAP a significant amount. However, when executing and evaluating this version, we noticed that the majority of errors generated by the model were surrounded by the classification of 6's and 9's. When deployed on a very simple test file, which consisted of using OpenCV to launch a continuous webcam stream for which the YOLOv8 model can be deployed on using the trained weights generated in Colab, the model continuously failed to distinguish between 6's and 9's. At this point, we decided to scrap the 9's classes from our dataset in hopes of improving the accuracy. We figured that this would be easier than devoting more time to increasing our dataset size. We knew that to reach a point in which our model would be comfortable distinguishing between 6's and 9's, a very very large number of training images would have needed to be added to the training set. Due to time restraints, remapping all the 9's classes to be identified as 6's was the best approach. From here on out, all version improvements were achieved by toying with the underrepresented class sets, trying to add as many images as possible so as to balance out the class instances.

Other Model Difficulties Encountered:

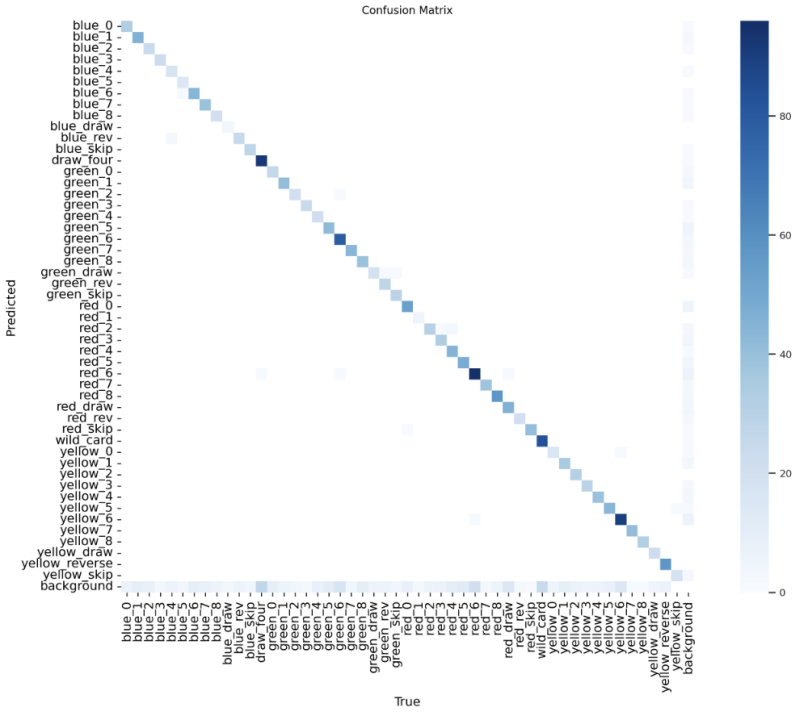
Overall, the model picked up on the specific card colors with amazing accuracy. However, sometimes it did struggle with the distinction between blue and green, especially in situations with bad lighting.

The issue with distinguishing between 6's and 9's was not an isolated one. We also noted that our system struggled with distinguishing between 7's and 1's, especially between blue/green cards. The system also struggled distinguishing between reverse cards and 1's. The reverse issue was easily solved by including a few more reverse cards in each color. However, the issue between the 7's and the 1's remained a persistent issue that carried on through our final versions.

Most of these issues could have been solved with a modification of the UNO game, such as playing with fewer numbers, eliminating the action cards, or reducing the colors involved in the gameplay. However, we were both determined to reach a point in our model where the bot could be incorporated into the gameplay in a faithful manner that adhered to the original rules of UNO. While this stubbornness possibly cost us the development of an extremely accurate demonstration, we believe that the majority of the faults of our UNO bot actually stem from the complexity of the model deployment and not the model itself. When tested independently of any additional logic computations on our simple webcam set up, both project members were very satisfied with the outcome produced by the final version. During three rounds of independent testing, version 9 recognized all the cards presented to it during a live webcam demo.

Final Training:

Final test images and their corresponding classifications



Confusion matrix for remapped classes

Processing and Preparing Input

The algorithms were simultaneously developed alongside the model's formulation. Presented below are provisional iterations of the algorithms, which would be employed in subsequent stages of the project. During this particular phase, while the AI model was still undergoing development, a representative dictionary was utilized to simulate the existence of input data that would typically be derived from the AI's analysis of the video feed.

This code defines a card game's card attributes and generates a sample hand and a sample deck.

The card attributes bank includes lists for colors, numbers, and actions. The colors list contains 'red', 'blue', 'green', 'yellow', and 'wild'. The numbers list contains '0' to '9', and the actions list contains 'skip', 'reverse', 'draw2', and 'draw4'.

The sample hand generator creates a dictionary called 'hand' to represent a player's hand. It iterates 7 times to generate 7 cards. For each iteration, it randomly selects a color from the colors list. If the color is 'wild', it randomly chooses between an action ('draw4') or no action (None), and the number is set to None. Otherwise, it randomly decides whether to assign a number or an action to the card. If it's a number, it randomly selects one from the numbers list,

and the action is set to None. If it's an action, the number is set to None, and it randomly chooses one from the actions list. The card is then added to the 'hand' dictionary with a key of 'CARD' followed by the current iteration number.

The hardcoded sample deck is a list called 'deck' that contains three elements: 'red', None, and 'skip'.

At this stage of the project, the subsequent code was devised to generate a sample Uno hand to facilitate the development of the algorithms:

```python
# CARD ATTRIBUTES BANK
colors = ['red', 'blue', 'green', 'yellow', 'wild']
numbers = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
actions = ['skip', 'reverse', 'draw2', 'draw4']


# SAMPLE HAND GENERATOR
hand = {}
for i in range(7):
    color = random.choice(colors)

    if color == 'wild':
        action = random.choice([None, 'draw4'])
        number = None
    else:
        is_number = random.randint(0, 1)  # randomly number or action

        if is_number:
            number = random.choice(numbers)
            action = None
        else:
            number = None
            action = random.choice(actions)

    card_name = 'CARD' + str(i+1)
    hand[card_name] = [color, number, action]

# HARDCODED SAMPLE DECK
deck = ['red', None, 'skip']
```

The following is the output that would be displayed in the terminal upon executing the aforementioned code. It is worth noting the specific format of the data associated with each card:

```
CARD1 ['blue', None, 'draw2']
CARD2 ['blue', None, 'reverse']
CARD3 ['green', None, 'draw2']
CARD4 ['yellow', '7', None]
CARD5 ['wild', None, 'draw4']
CARD6 ['blue', '2', None]
CARD7 ['blue', '8', None]
```

The AI model used the following classes to identify cards, as mentioned earlier. These strings were parsed out of the model's predictions for each card:

blue_0
blue_1
blue_2
blue_3
blue_4
blue_5
blue_6
blue_7
blue_8
blue_draw
blue_rev
blue_skip
draw_four
green_0
green_1
green_2
green_3
green_4
green_5
green_6
green_7
green_8
green_draw
green_rev
green_skip
red_0
red_1
red_2
red_3
red_4
red_5
red_6
red_7
red_8
red_draw
red_rev
red_skip
wild_card
yellow_0
yellow_1
yellow_2
yellow_3
yellow_4
yellow_5
yellow_6
yellow_7
yellow_8
yellow_draw
yellow_rev
yellow_skip

To transform such strings into the format mentioned in the sample hand generator, the following function was employed. The rationale behind this function was to ensure that the logic algorithm, which will be elaborated upon subsequently, processed the specific three-item array value rather than the raw string output generated by the AI model.

```python
def card_to_array(card):
    # Initialize the array with None values
    arr = [None, None, None]

    # Handle special cases first
    if card == "draw_four":
        return ['wild', None, 'draw4']
    if card == "wild_card":
        return ['wild', None, None]
```

```
# For other cases, split the card string by "_"
parts = card.split("_")

# The first part is the color
arr[0] = parts[0]

# The second part is either a number or an action
if parts[1].isdigit():
    arr[1] = parts[1]
else:
    arr[1] = None
    if parts[1] == "rev":
        arr[2] = "reverse"
    elif parts[1] == "skip":
        arr[2] = "skip"
    elif parts[1] == "draw":
        arr[2] = "draw2"

return arr
```

Logic Development

We wanted to give our system two tiers of functionality:

1. Play to Play
2. Play to Win

The first approach enables a player to participate in the game without a specific objective of winning, focusing instead on following the rules and actively engaging in the gameplay. On the other hand, the second approach is geared towards optimizing the chances of winning against a human opponent, prioritizing strategic moves that yield a significant success rate.

Play to Play

In the Play to Play algorithm, the system employs a straightforward approach to select a card from the deck that allows for the play of any legally permissible card, without any external factors influencing the decision-making process. In the given code snippet, the algorithm selects the very first card from the array of possible cards based on this criterion.

The function get_possible_hand is responsible for identifying the playable cards within a given hand, taking into consideration the current top card of the deck. The function iterates through

each card in the hand, evaluating various conditions to determine if a card can be played. If the top card of the deck is a "wild" card, any card in the hand is considered playable. If it is a "draw2" card, the function immediately returns "draw2," indicating that the player should draw two cards. Similarly, if the top card is a "skip" card, the function returns "skip," denoting that the player's turn should be skipped.

Furthermore, the function checks for matches in terms of color, number (excluding None), action (excluding None), and "wild" cards. If no viable cards can be played based on these criteria, the function returns None. However, if there are playable cards available, the function selects the first viable card and designates it as the chosen card, subsequently returning it as the output of the function.

```python
def get_possible_hand(hand, deck):
    color = deck[0]
    number = deck[1]
    action = deck[2]
    possible_hand = []

    for card in hand:
        card_color = card[0]
        card_number = card[1]
        card_action = card[2]
        if color == 'wild':
            possible_hand.append(card)
        elif action == 'draw2':
            return 'draw2'
        elif action == 'skip':
            return 'skip'
        elif card_color == color or (card_number == number and card_number != None) or (
                card_action == action and card_action != None) or card_color == "wild":
            possible_hand.append(card)

    if not possible_hand:  # Check if the list is empty
        return None

    choice = possible_hand[0]
    return choice
```

Play to Win

The Play to Win algorithm would theoretically incorporate either heuristics or probability insights to determine the optimal card to play, adhering to the rules of Uno while leveraging mathematical principles to guide the user of the system towards achieving a victory in a more expeditious manner.

Leveraging Heuristics

Below is the algorithm we've devised that will operate on a Heuristic Search algorithm. Heuristic Search involves using domain-specific knowledge to guide the search for a solution, which can be effective in games like Uno that have simple rules and a limited number of possible moves. By using rules of thumb such as playing high-numbered cards early in the game, or avoiding playing Wild Draw Four cards unless necessary, Heuristic Search can quickly identify the best moves to make in a given game state.

The following heuristics were used in this function. These heuristics were chosen because they prioritize aggressive gameplay, focusing on disrupting the opponent's turn and controlling the game's flow:

1. **If there is an action card of the same color as the deck card, play it.** Action cards like "reverse", "skip", and "draw two" can disrupt the opponent's turn and delay their chances of winning. By prioritizing these cards, the player can control the flow of the game and create a more favorable situation for themselves.
2. **If there is a wildcard, play it.** Wildcards like "wild" and "wild draw four" cards can be played on any card, and they allow the player to choose the color they want to play next. This can help the player adapt to different situations and potentially force the opponent to draw more cards, making it harder for them to win.
3. **If there is a card with the same color as the deck card, play it.** Playing cards with the same color as the deck card increases the chances of the opponent not having a matching card to play, forcing them to draw more cards or miss their turn.
4. **If there is a card with the same number as the deck card, play it.** If none of the above options are available, playing a card with the same number as the deck card is still a valid move. While it doesn't provide any direct advantage over the opponent, it allows the player to reduce the number of cards in their hand, bringing them closer to winning the game.

Four lists are created to store cards that match the heuristics: same_color_cards, same_number_cards, same_color_action_cards, and wildcards. The color, number, and action of the deck card are then unpacked into the variables deck_color, deck_number, and deck_action.

The function iterates through the cards in the player's hand using a for loop. For each card, it unpacks the color, number, and action into the variables card_color, card_number, and card_action. The function checks if the card's color and action match the deck's color and if the card has an action. If this condition is met, the card's name is appended to the same_color_action_cards list. If the card is a wildcard, its name is added to the wildcards list. The function also checks if the card's color matches the deck's color and if the card has a

number, in which case it appends the card's name to the same_color_cards list. If the card's number matches the deck's number and the card has a number, the card's name is appended to the same_number_cards list.

After checking all the cards in the hand, the function returns the best card to play based on the heuristics. If a list is not empty, the first card from the list is returned. The priority order is: same_color_action_cards, wildcards, same_color_cards, and same_number_cards. If none of these lists have any cards, the function returns None, indicating that no valid card can be played.

Python

```python
def best_card_to_play(hand, deck):
    same_color_cards = []
    same_number_cards = []
    same_color_action_cards = []
    wildcards = []

    deck_color, deck_number, deck_action = deck

    for card_name, card_info in hand.items():
        card_color, card_number, card_action = card_info

        # Heuristic 3: Check for same color action cards
        if card_color == deck_color and card_action is not None:
            same_color_action_cards.append(card_name)

        # Heuristic 4: Check for wildcards
        if card_color is None and card_action is not None:
            wildcards.append(card_name)

        # Heuristic 1: Check for same color
        if card_color == deck_color and card_number is not None:
            same_color_cards.append(card_name)

        # Heuristic 2: Check for same number
        if card_number == deck_number and card_number is not None:
            same_number_cards.append(card_name)

    # Return the best card to play based on heuristics
    if same_color_action_cards:
        return same_color_action_cards[0]
    elif wildcards:
        return wildcards[0]
    elif same_color_cards:
        return same_color_cards[0]
    elif same_number_cards:
        return same_number_cards[0]
    else:
        return None
```

Leveraging Probability

The strategy that the function below employs is based on the idea that by playing the card of the color that appears least frequently in the player's hand and memory, the player is more likely to force their opponent to draw cards or not be able to play their cards, increasing the player's chances of winning.

The calculate_best_card() function is used to determine the best card to play for player 2 given the current state of the game. It is designed to choose the card based on the color frequency in both the player's hand and the memory (cards that have been played).

Here's a step-by-step breakdown of what the function does:

1. **Color counting:** The function first initializes a dictionary color_count to keep track of the number of cards of each color ("red", "green", "blue", "yellow") in the player's hand and the memory. The function then iterates over all the cards in the hand and the memory, and for each non-wild card, it increments the corresponding color count in the dictionary.
2. **Finding the best card:** The function then initializes two variables, min_color_count and best_card, to keep track of the color with the lowest count and the best card to play, respectively. It then iterates over all playable cards. For each non-wild card, it checks if its color count is less than min_color_count. If it is, it updates min_color_count and best_card. The result is that best_card will be the card whose color appears least frequently in the player's hand and memory.
3. **Wild card handling:** If no non-wild card is found in the previous step, the function then iterates over all playable cards again to find a wild card. If it finds one, it sets it as the best card to play.
4. The function finally returns the best_card.

```python
def calculate_best_card(hand, playable_cards, memory):
    # Basic strategy: Choose the card with the color that appears least in memory and hand
    color_count = {"red": 0, "green": 0, "blue": 0, "yellow": 0}
    for card in hand + memory:
        if card[0] != "wild":
            color_count[card[0]] += 1

    min_color_count = float('inf')
    best_card = None
```

```
    for card in playable_cards:
        if card[0] == "wild":
            # Wild cards should be considered last
            continue
        card_color_count = color_count[card[0]]
        if card_color_count < min_color_count:
            min_color_count = card_color_count
            best_card = card

    # If no non-wild card is found, choose a wild card if available
    if best_card is None:
        for card in playable_cards:
            if card[0] == "wild":
                best_card = card
                break

    return best_card
```

Simulated Testing

To assess the effectiveness of each algorithm, functions were implemented to simulate gameplay. One function emulated a player utilizing one algorithm, while the other function simulated a player employing another. These functions were used to play multiple rounds, tracking the number of wins for each algorithm. By comparing the win rates of the two algorithms, the relative effectiveness of each approach could be evaluated.

The simulate_gameplay_once(function1, function2) simulates a game of UNO between two players and serves as a helper function to simulate_gameplay_multiple_times(num_plays, func1, func2). The players' strategies are determined by the functions function1 and function2 passed as parameters to simulate_gameplay_once.

1. **Set up the deck of cards:** It first creates a deck of UNO cards by looping through the possible colors, numbers, and action types. It assigns each card a unique identifier (CARD1, CARD2, etc.) and stores the card attributes in a dictionary, where the key is the card's identifier and the value is a list of the card's attributes.
2. **Initialize the players' hands and the deck:** It randomly selects a card from the deck to be the first card on the table (the "deck" in the context of the game rules). It also generates the players' hands using the generate_hand function (which isn't defined in the provided code).
3. **Gameplay loop:** The game continues in a loop until a player has no more cards left in their hand (which means they've won the game).

    a. On a player's turn, they play a card from their hand onto the deck using their strategy function (function1 for Player 1 and function2 for Player 2). The strategy function is assumed to return the identifier of the card to play.
    b. If a player can't play a card (their strategy function returns None), they draw a random card from the initial card set.
    c. After each turn, it checks if the player's hand is empty. If it is, it sets the game_over flag to True to end the game loop.
4. **Determine the winner:** After the game loop ends, the function determines the winner by checking which player has no cards left in their hand. It returns a string indicating the winner ("Player 1" or "Player 2").

In this simulation, the card rules (like the effects of action cards) aren't implemented. This is a simplified version of UNO gameplay, where the primary goal is to get rid of all your cards, and the strategy functions decide which card to play.

Python

```python
def simulate_gameplay_once(function1, function2):

    # CARD ATTRIBUTES BANK
    colors = ['red', 'blue', 'green', 'yellow', 'wild']
    numbers = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
    actions = ['skip', 'reverse', 'draw2', 'draw4']

    # GENERATE ALL POSSIBLE UNO CARDS
    all_cards = {}
    card_id = 1
    for color in colors[:-1]:  # Exclude 'wild' from colors
        for number in numbers:
            for _ in range(2):  # Add each number card twice
                card_name = f"CARD{card_id}"
                all_cards[card_name] = [color, number, None]
                card_id += 1

        for action in actions[:-1]:  # Exclude 'draw4' from actions
            for _ in range(2):  # Add each action card twice
                card_name = f"CARD{card_id}"
                all_cards[card_name] = [color, None, action]
                card_id += 1

    for _ in range(4):  # Add wild and wild draw4 cards
        card_name = f"CARD{card_id}"
        all_cards[card_name] = ["wild", None, None]
        card_id += 1
        card_name = f"CARD{card_id}"
        all_cards[card_name] = ["wild", None, "draw4"]
        card_id += 1
```

```
# INITIALIZE DECK WITH A RANDOM CARD
deck = random.choice(list(all_cards.values()))

player1_hand = generate_hand(all_cards)
player2_hand = generate_hand(all_cards)
deck = random.choice(list(all_cards.values()))

player1_turns = 0
player2_turns = 0
game_over = False

while not game_over:
    # Player 1's turn
    card_to_play = function1(player1_hand, deck)
    if card_to_play:
        deck = player1_hand[card_to_play]
        del player1_hand[card_to_play]
        # print("Player 1 played:", card_to_play, deck)
        player1_turns += 1
    else:
        card_name, card_values = random.choice(list(all_cards.items()))
        player1_hand[card_name] = card_values

    if not player1_hand:
        game_over = True
        break

    # Player 2's turn
    card_to_play = function2(player2_hand, deck)
    if card_to_play:
        deck = player2_hand[card_to_play]
        del player2_hand[card_to_play]
        # print("Player 2 played:", card_to_play, deck)
        player2_turns += 1
    else:
        card_name, card_values = random.choice(list(all_cards.items()))
        player2_hand[card_name] = card_values

    if not player2_hand:
        game_over = True
        break

if not player1_hand:
    return "Player 1"
elif not player2_hand:
    return "Player 2"
```

The simulate_gameplay_multiple_times(num_plays, func1, func2) function is used to run multiple simulations of the UNO game using the simulate_gameplay_once(func1, func2) function defined earlier. A brief explanation of the function is given below:

1. **Initialize win counts:** It first initializes two variables, player1_wins and player2_wins, to count the number of games won by each player.
2. **Run simulations:** It then runs num_plays number of simulations by calling simulate_gameplay_once(func1, func2), which simulates a single game of UNO using the strategies defined by func1 and func2. After each game, it checks the winner and increments the corresponding player's win count.
3. **Print results:** After all the games have been simulated, it prints out the number of games won by each player.

This function was useful to evaluate and compare different strategies (represented by func1 and func2) over multiple games, as the results could differ due to the inherent randomness in the game (such as the order of cards in the deck and the cards drawn when a player can't play a card).

```python
def simulate_gameplay_multiple_times(num_plays, func1, func2):
    player1_wins = 0
    player2_wins = 0

    for i in range(num_plays):
        winner = simulate_gameplay_once(func1, func2)
        if winner == "Player 1":
            player1_wins += 1
        else:
            player2_wins += 1
    print(f"Player 1 wins: {player1_wins}")
    print(f"Player 2 wins: {player2_wins}")
```

Results & Decision on Final Algorithm

In this testing instance using simulate_gameplay_multiple_times(num_plays, func1, func2), player 1 adopted the algorithm associated with the Play to Play functionality, which focused on selecting cards that allowed for legal plays without strategic considerations for winning. Meanwhile, player 2 implemented the heuristics-based algorithm designed to maximize the likelihood of winning more frequently.

Player 1 wins: 834
Player 2 wins: 166

Player 1 wins: 899
Player 2 wins: 101

Player 1 wins: 847
Player 2 wins: 153

Player 1 wins: 882
Player 2 wins: 118

Player 1 wins: 815
Player 2 wins: 185

Player 1 wins: 848
Player 2 wins: 152

Player 1 wins: 869
Player 2 wins: 131

Unfortunately, the heuristic algorithm did not achieve the intended objective of increasing the success rate in winning the game. Despite the incorporation of heuristics and probability insights, it appears that the algorithm did not yield the desired outcome.

In this subsequent testing instance, the testing structure remains unchanged, but there is a notable modification: Player 2 now adopts the probability-based algorithm. Meanwhile, Player 1 continues to utilize the same algorithm as before.

Player 1 won 655 times.
Player 2 won 345 times.

Player 1 won 657 times.
Player 2 won 343 times.

Player 1 won 633 times.
Player 2 won 367 times.

Player 1 won 657 times.
Player 2 won 343 times.

Player 1 won 636 times.
Player 2 won 364 times.

Player 1 won 648 times.
Player 2 won 352 times.

In conclusion, while the heuristic and probability-based algorithms implemented by player 2 were designed to increase the chances of winning, they did not achieve the expected results in practice. It seems counterintuitive, but the simple Play to Play algorithm used by player 1 consistently outperformed both of these more sophisticated strategies, which had been predicted to lead to a higher win rate. This paradox might suggest that the inherent complexity and unpredictability of the game can sometimes defy even well-reasoned strategic models.

Perhaps the Play to Play algorithm, by focusing on legal plays without considering strategies for winning, managed to incorporate a form of 'randomness' that proved advantageous in this unpredictable game environment. These findings led us to implement solely the Play to Play algorithm in our code, as it consistently yielded better results than the other two strategies, despite their theoretical advantages.

This testing underscores the importance of empirical testing and the potential limitations of relying solely on strategic modeling when designing game-playing algorithms. It also highlights the intriguing possibility that sometimes, a simpler approach can outperform more complex strategies.

Human Testing

We brought in a human player to compete against each algorithm to test their performance against human gameplay behavior. The human player, who we'll refer to as "John," faced off against the Play to Play, Heuristic, and Probability algorithms.

In the first instance, John was pitted against the Play to Play algorithm. The data from these games is as follows:

Play to Play wins: 8
John's wins: 2

Play to Play wins: 9
John's wins: 1

The Play to Play algorithm consistently outperformed John in each of these rounds. Next, John went up against the Heuristic algorithm:

Heuristic wins: 3
John's wins: 7

Heuristic wins: 2

John's wins: 8

The Heuristic algorithm was unable to beat John consistently, falling short in each round. Finally, John played against the Probability algorithm:

Probability wins: 4
John's wins: 6

Probability wins: 3
John's wins: 7

While the Probability algorithm performed slightly better than the Heuristic algorithm, it still couldn't match the success rate of the Play to Play algorithm.

Thus, despite the human element and the theoretical advantages of the Heuristic and Probability algorithms, the Play to Play algorithm once again proved superior. These results further confirm our decision to implement solely the Play to Play algorithm in our code.

Model Deployment

To deploy our model in a way that would help us integrate it seamlessly into a live game of uno, we knew that we needed two forms of visual input: one that read the live deck and one that read the player’s hand. While we first suggested a one camera system, we found the domain engineering of this quite difficult since all players are supposed to be able to see the live deck, but players are not allowed to see each other's (card) hands. However, Geneva’s computer comes equipped with two cameras: a traditional webcam placed on the top center of the computer screen and another webcam on the back facing panel of the computer. After testing it out, we confirmed that both of these video streams could be launched simultaneously.

Once we decided on this design choice, we then had to construct our model deployment and game logic around this domain engineering. The back camera would read in the live deck, a card placed on a black platform that was tilted toward the camera at an angle of 45 degrees. The front camera would act as a traditional webcam, with the player turning their fanned hand toward the camera.

Initially, the subsequent code snippet was employed to implement the comprehensive system that encompassed the developed model and the associated logic:

It begins by tying the model into our code via Roboflow's API. Next, it opens two cameras using the cv2.VideoCapture() function. The hand and deck arrays are initialized which are later filled with the results of predictions made on the captured frames.

The main function of the program is contained within a *with* statement. It creates a ThreadPoolExecutor, which is a high-level interface for asynchronously executing callables. This was used in an attempt to parallelize these two tasks to enhance camera performance. This feature effectively improved the overall camera quality, which was previously characterized by choppiness.

Inside the *while True:* loop, it cycles through the two cameras, captures a frame from each, then applies a prediction to the frame. If a frame was successfully captured, it displays the frame in an OpenCV window. It exits the loop (and therefore the program) if the ESC key is pressed.

Once the loop is exited, it processes the hand and deck arrays into a new format, then prints them out. It then makes a decision based on the recognized cards and prints out either a suggestion to draw another card or a suggestion of which card to play.

```python
#CONNECT TO API
rf = Roboflow(api_key="IwzWQhIef61ofB1J8Q4F")
project = rf.workspace().project("uno-card-bot")
model = project.version(9).model

# OPEN CAMERAS
cap0 = cv2.VideoCapture(0)
cap1 = cv2.VideoCapture(1)

#INITIALIZE ARRAYS
hand = []
deck = []

# MAIN FUNC - APPLIES PREDICTION FUNCTION TO FRAMES
with concurrent.futures.ThreadPoolExecutor() as executor:
    while True:
        for i, cap in enumerate([cap0, cap1]):
            future_frame = executor.submit(capture_frame, cap)
            frame = future_frame.result()
            if frame is not None:
                executor.submit(predict, frame, i)

            # Display the frame to the user
            cv2.imshow(f'Input {i}', frame)

        # Exit if ESC key is pressed
```

```
        if cv2.waitKey(2) & 0xFF == 27:
            break

# CONVERT HAND TO FUNCTION-READY FORMAT
hand = [card_to_array(card) for card in hand]
deck = deck[0] #chosing only the first prediction on the deck as the official prediction on the deck
deck = card_to_array(deck)

# CONFIRM HAND AND DECK
print("Hand", hand)
print("Deck", deck)

# PRINT RESULTS
result = get_possible_hand(hand, deck)
if not result:
    print("draw another card!")
else:
    print("play this card: ", result)
```

Issues in Implementation

As mentioned in the Model Training section, when independently tested, our object detection model ran with much success. However, when the video feed that was introduced into the model was slow, glitchy, and not continuous, the model had great difficulty detecting the cards that were presented before it. Whenever we tried to run this live, two camara simultaneous model, the slowness of the webcam video feed failed to faithfully reproduce the visual input needed for our model to work correctly.

Regarding the issue of slowness, the main bottleneck seems to be that the frames are being processed sequentially due to the future_frame.result() call. This is a blocking operation which waits for the frame to be captured before it continues, which could have been the culprit for why the entire loop was slowed down, especially when the capture inputs were provided simultaneously.

Though we did implement the concurrent.futures.ThreadPoolExecutor(), further research showed that, as implemented, the blocking result() call was likely negating much of the benefit of multithreading.

In another attempt, we tried to divide the video input between two machines as it seemed that perhaps launching two cameras on the same device could have been the issue creating the

lag. We attempted to run the deck camara on Geneva's computer and the hand camera on Luciana's computer, communicating the results through a server/client socket connection. However, this was not successful.

After troubleshooting unsuccessfully, we understood that we had to abandon our attempt at a live video feed.

Solution

Our solution was to separate the capture and prediction steps which helped to solve the performance issue that was seen in the previous code provided. In the previous code, capturing from the cameras and predicting using the machine learning model were happening concurrently within the same loop, which led to slowdowns, particularly because the model prediction was time-consuming.

In the code below:

1. The main loop is dedicated to capturing and displaying the images from the cameras. This loop runs indefinitely until the user either saves the current frames with 's' key or quits the program with 'q' key.
2. After an image is saved or the program is ended (and thus the cameras are released), the code then loads a YOLO model and uses it to predict the contents of the saved images.
3. The results of these predictions (which are expected to be the detected Uno cards) are then processed and printed out.

```python
while True:
    try:
        check1, frame1 = deckcam.read()
        check2, frame2 = selfiecam.read()

        cv2.imshow("Capturing 1", frame1)
        cv2.imshow("Capturing 2", frame2)

        key = cv2.waitKey(1)
        if key == ord('s'):
            cv2.imwrite(filename='deck.jpg', img=frame1)
            cv2.imwrite(filename='hand.jpg', img=frame2)
            deckcam.release()
            selfiecam.release()
            cv2.waitKey(1650)
```

```
            cv2.destroyAllWindows()
            print("Image saved!")
            break

        elif key == ord('q'):
            print("Turning off camera.")
            deckcam.release()
            selfiecam.release()
            print("Camera off.")
            print("Program ended.")
            cv2.destroyAllWindows()
            break

    except(KeyboardInterrupt):
        print("Turning off camera.")
        deckcam.release()
        selfiecam.release()
        print("Program ended.")
        cv2.destroyAllWindows()
        break


# Load the local YOLO model
model = YOLO("best_2.pt")

# Class names
classNames = ["blue_0","blue_1","blue_2", "blue_3", "blue_4", "blue_5", "blue_6", "blue_7", "blue_8", "blue_draw",
"blue_rev", "blue_skip", "draw_four", "green_0","green_1","green_2", "green_3", "green_4", "green_5", "green_6",
"green_7", "green_8", "green_draw", "green_rev", "green_skip", "red_0","red_1","red_2", "red_3", "red_4", "red_5",
"red_6", "red_7", "red_8", "red_draw", "red_rev", "red_skip", "wild_card", "yellow_0","yellow_1","yellow_2", "yellow_3",
"yellow_4", "yellow_5", "yellow_6", "yellow_7", "yellow_8", "yellow_draw", "yellow_rev", "yellow_skip"]

deck_results = model.predict("deck.jpg")
hand_results = model.predict("hand.jpg")

deck_raw = []
hand_raw = []

for r in deck_results:
    boxes = r.boxes
    for box in boxes:
        cls = int(box.cls[0])
        cardDetected = classNames[cls]
        # print(cardDetected)
        deck_raw.append(cardDetected)

for r in hand_results:
    boxes = r.boxes
    for box in boxes:
        cls = int(box.cls[0])
        cardDetected = classNames[cls]
        # print(cardDetected)
```

```
        hand_raw.append(cardDetected)


deck = deck_raw[0]
hand = hand_raw

deck = card_to_array(deck)
hand = [card_to_array(card) for card in hand]

print("hand", hand)
print("deck", deck)

# PRINT RESULTS
result = get_possible_hand(hand, deck)
if result == 'skip':
    print('sorry, turn skipped')
elif result == 'draw2':
    print("sorry, draw 2 cards")
elif not result:
    print("draw another card!")
else:
    print("put down this card: ", result)
```

By separating the capture and prediction steps into different parts of the program, the capture loop can run as fast as possible without being slowed down by the model prediction. The predictions are only made after the user decides to save the current frames, which avoids the need to make predictions on every frame.

This structure also means that the potentially slow prediction step doesn't block the capture loop, because the predictions are only made after the loop has finished. This allows the capture loop to run at near the maximum frame rate supported by your cameras, improving the real-time performance of the capture step.

The prediction step only happens once per user command, rather than on every frame, which can significantly reduce the computational load of this part of the program. This might be particularly beneficial if the model prediction is computationally intensive, as is often the case with trained models like YOLO.

Demonstration

The functionality of the system was demonstrated through the utilization of Geneva's dual-camera laptop, the system on which the script was loaded. To ensure accurate card reading, an inactive iPad was positioned in front of the deck-reading camera, serving as a high-contrast black background upon which the deck card could be placed (the black platform

that was referred to earlier). The card, which initiated the commencement of the game, was affixed to the screen in a manner that ensured it was squarely positioned before the deck-reading camera.

During gameplay, the player held a fan of cards consisting of three to six cards, with the numbers on the left-hand corner of each card being visible. This fan of cards was observable by the user-facing camera, which captured the hand currently in play.

Upon execution of the script, the user activated the designated key to capture visual feeds from both cameras. The system's output, which provided an overview of the current hand, the present deck, and the suggested move, was then displayed in the terminal.

Subsequently, the player made a selection of a card to play based on the system’s suggestion, replacing the previously mounted deck card with the chosen one. This process was repeated iteratively until the game reached its conclusion. Between these turns, the other participant would engage in regular gameplay without the assistance of the computer, following the standard protocol of replacing the deck card. The players would alternate turns in accordance with the conventional gameplay rules of Uno.

Reflections

Ultimately, while our model worked very successfully when implemented independent of any logic processes, it was its incorporation into the broader game system that caused many problems for us. The following solutions are proposed to possibly increase the accuracy of our demonstration itself

1. Increase the models accuracy to the point where a delayed video feed does not interfere so severely with the class predictions
2. Run the video inputs on independent machines and find a manner to communicate the live video results to a single machine that takes care of the expensive logic computations
3. Utilize a high resolution camera capable
4. Utilize an external second camera
5. Improve the lighting conditions

We believe that many of these suggestions could have been better implemented into our system, if we were not limited by the time constraints. However, we did not predict to run into so many difficulties during the model training process. This was sacrificed time that could have gone towards the implementation of our model. Nevertheless, this project was an amazing opportunity to delve into an unknown aspect of computer vision for both of us, and we have learned a tremendous amount during the month it took to develop this.