ECE 113D Final Project: SET Classifier

Tyler Price, Bradley Schulz

Abstract — In this project we apply computer vision techniques to the card game SET in which players must identify groups of complementary cards based on 4 visual attributes. With this, we aimed to gain experience with computer vision algorithms, especially in the context of embedded systems, through working to find a technological approach to a common game played among friends. We developed algorithms to classify each attribute of the cards — number of shapes, type of shape, shape color, and shape fill — all with the goal of being as computationally lightweight as possible. In the end, we created a system that is approximately 86% effective and can identify valid sets of cards in under 5 seconds.

I. Introduction

A. History

Computer vision is the process by which computers aim to mimic humans' natural vision systems to be able to classify and localize objects. This field has existed since 1960 when Larry Roberts at MIT began theorizing how to extract 3-dimensional information from a 2-dimensional image [1]. There are numerous applications for computer vision including automation of manufacturing tasks, autonomous driving, and inspection of finished products [1].

Computer vision tasks can involve one or either of two tasks: classification—determining to which pre-defined class an image belongs—and localization—the process of identifying the location of one or more distinct objects within an image [2]. For tasks that require both localization and classification, there are many algorithms that exist to both localize and classify multiple objects within a single image such as RCNN [3], SIFT [4], and YOLO [2]. However, these algorithms are often too computationally expensive for smaller scale embedded systems, and much research is currently exploring how to simplify these complex tasks to work on real-time embedded systems [5] [6] [7]. Since embedded systems are subject to often significant power and performance constraints due to their small size and lack of a constant source of power, it is vital that any image processing tasks operating on an embedded system are as efficient as possible. In many cases, one must find an algorithm that is optimized for the specific task at hand so that it most efficiently uses the hardware resources available.

In this project, we apply computer vision to the card game SET. This game was invented in 1974 by Marsha Jean Falco while studying the genetics of dogs [8]. The goal of the game is for players to identify sets of 3 cards as fast as possible. There are 81 cards, each with 4 attributes: color, shape type, number of shapes, and shape fill. There are 3 variations of each of the 4 attributes for a total of $3^4 = 81$ total cards. Table I shows the possible attributes.

TABLE I
SET Card Attribute Descriptions

Attribute			
Shape Type	Oval	Diamond	Squiggle
Shape Color	Red	Green	Purple
Shape Fill	Empty	Striped	Solid
Number of Shapes	1	2	3

A valid set is one in which each attribute of each of the three cards are either all the same or all identical. For example, the following cards are valid sets.

TABLE II
Example Valid Set

Card	Shape Type	Shape Color	Shape Fill	# of Shapes
1	Oval	Red	Solid	2
2	Oval	Purple	Empty	2
3	Oval	Green	Striped	2

Although this card game may appear simple, it presents many interesting problems relating to combinatorics, error-correcting codes, and Fourier analysis [8].

B. Global Constraints

Working with embedded systems comes with several inherent constraints, as the computing resources on small devices are often significantly limited. All power must come from a battery, so increases in power consumption require either a larger battery or more frequent recharging. The number of computations should also be minimized, since increasing the computational demand requires either a larger processor or more power. There is room for optimization in both hardware and software, but these power and computational constraints never go away.

In our context, we are limited by the hardware available to us in our immediate context. We were provided the OpenMV Cam H7 which contains an STM32 microcontroller and OV7725 camera module. Both of these hardware components have limits in computational power and camera resolution, respectively. Moreover, the software libraries available on the OpenMV IDE used for this device are much more limited than what is available on other Python distributions. Standard Python libraries such as Numpy and OpenCV have optimized functions that use code written in C to speed up their execution. However, since these two libraries are not available on the OpenMV IDE, any replacement function will not be as fast because we can only write Python-based substitutions in this IDE. Thus, we don't have the capabilities to implement the optimized C code that would make these algorithms run more efficiently on the available hardware.

II. MOTIVATION

In our prior research, we found a few other projects that solved the game SET using real time image processing, although none of these were intended to be used on embedded systems [9] [10] [11]. Some focused solely on the algorithm side of the problem with the assumption that the code would be running on a system with large computational resources and access to the OpenCV Python library [9] [10]. One system we found that considered hardware targeted the code to run on a mobile phone as a standalone app, which again has plenty of computational resources [11]. Regardless, none of the documented attempts we found to make a system to solve the game SET were designed to be lightweight algorithms that can run on a low-power device. We wanted to design something that could operate on its own that does not require complex hardware and can be inconspicuously held within your hand.

We also chose this task because we wanted to gain experience with embedded image processing. With the rising prevalence of computer vision in applications such as autonomous vehicles, manufacturing, security, and even agriculture, we wanted to gain experience in this technology which will likely be at the forefront of technology in the coming years [12]. We had some theoretical knowledge of the concepts behind computer vision, but no hands-on experience implementing these tools ourselves. We had worked with some baseline computer vision functions in Python, but never beyond class projects.

Moreover, having to perform image processing on an embedded system is an excellent opportunity to learn these skills while aiming to keep the system as simple as possible. Simpler solutions that use fewer computational resources are almost always preferable, and using an embedded system forces us to work with resource constraints that arise in real life engineering. In the future, we hope to apply the skills learned in this project to real world projects requiring efficient image processing in applications such as edge IoT devices.

Lastly, we enjoy playing SET with friends, and we wanted to work on a project with personal significance. We aimed to be able to use this system in real gameplay to win actual games of SET.

III. APPROACH

A. Team Organization

Our team consisted of two people: Tyler Price and Bradley Schulz. We both have experience working with embedded systems and real-time processing, but our experience with computer vision differed slightly. Bradley has taken more coursework on machine learning, computer vision, and Python, so he focused more on developing and testing the algorithms before implementing them on the final board. Tyler focused more on the hardware side by adapting the algorithms developed by Bradley to run on the final board. This involved several changes to the algorithms based on the constraints arising from real-life implementation. We met every week to discuss our progress and collaborate on any issues that arose.

B. Plan and Implementation

We had 10 weeks to develop our project, and we set the third and seventh weeks as major milestones for our progress. For the third week, our goals were to be able to read images from a camera on the H7 and have an algorithm that could classify an image of a card. By week 7, we aimed to expand that algorithm to cover both segmenting the raw image into individual images of each of the 12 cards in play and classifying each card in the image. On the hardware side, we also aimed to have a working LED matrix by week 7 that would be our main user interface.

We successfully met both major milestones. Reading data from the camera went smoother than we intended, and we were able to get this working by the end of the first week. For classifying an image of a single card, we were able to get this working on Google Colab by the third week. It took some more work than we intended to port our algorithms from Google Colab (where we developed them) to the H7 due to differences between the standard Python language and MicroPython which runs on the H7, but we were still able to complete the transition by week 7. We had to make some significant changes to the algorithms, but we hit our week 7 milestone nonetheless.

The matrix was also significantly easier than we anticipated to get working, and we had it up and running by week 4. We decided to use a pre-made matrix which simplified both hardware and software development thanks to pre-made I2C libraries built for the matrix.

However, there were several challenges we encountered along the way. Thresholding and pre-processing took a significant amount of trial and error to perfect. OpenMV's automatic thresholding function performed inconsistently across different lighting conditions and shape colors, and glare would cause pixels from the card background to pass through thresholding. This severely impacted the reliability of our original shape counting algorithm which relied on isolating pixels corresponding to the shapes. We overcame the effects of glare by pivoting to an edge-based shape counting algorithm.

We also encountered challenges due to the differences in the environment we used to develop the algorithms on Google Colab and real-life execution. Initially, we used a pre-made training set of images we found on Github to train our classification algorithms. However, the images collected by the OV7725 camera on the OpenMV board were of much lower resolution and had more noise than these training images. Since some of our algorithms relied on empirically determined thresholds and a custom-trained CNN, our algorithms became significantly less accurate when we first ported them to the H7. We overcame these obstacles by generating our own custom training data and redeveloping our malfunctioning algorithms.

Additionally, not all the library functions we used on Google Colab were available on OpenMV. We used OpenCV and Numpy on Google Colab since we were familiar with these libraries and using them made it very fast to develop and test algorithms. We assumed that equivalent operations would be available on OpenMV, but there were times when there were not. This was the case in our color and fill classification algorithms which used OpenCV to isolate shapes and NumPy to analyze color ratios through their standard deviation. We ended up finding more efficient algorithms that analyzed the colors in the LAB color space instead of the RGB color space like our previous algorithms.

A full outline of our weekly goals is in Appendix A. We successfully accomplished all our weekly goals with the exception of week 6 due to issues implementing the functions from Google Colab in OpenMV.

C. Standard

For most computer vision tasks, OpenCV is the go-to code library. Even competing software libraries recommend OpenCV because it is easy to work with and can be used with languages such as Python, C++, MATLAB, Java, and more [13] [14] [15]. This library contains numerous pre-made classes and functions that remove the need to re-implement the algorithms such as blurring, filtering, thresholding, and even object detection. Using the OpenCV library makes it very easy to develop and test algorithms.

However, OpenCV is not a standard library for embedded hardware. Many embedded systems that run Python use a lighter-weight version called MicroPython or one of its derivations such as Adafruit's CircuitPython [16]. MicroPython implements a pared-down version of the main Python language so that it runs smoothly on embedded systems where computing resources are more limited. This means that it does not have access to all the libraries available on a full Python distribution; it only contains the most important and applicable ones [16].

To communicate with the LED matrix, we use the I2C (or Inter Integrated Circuit) standard for communication. This standard was developed by Philips Semiconductors and is currently in version 6.0 [17]. It uses a four wire interface with a power, ground, data, and clock line that can connect multiple devices all in parallel. It is used in many inter-board applications where devices need to communicate with each other without many wires. We chose to use it because it is a well-developed standard and is supported by the hardware we used.

D. Theory

Although the computer vision functions we used from OpenCV and OpenMV implement all the computer vision algorithms for us, it is important to understand what computations are occurring behind the scenes. One reason for this is that knowing which algorithms are more computationally expensive than others allows us to choose algorithms that minimize the complexity of our system and increase the efficiency of our classification.

Convolution and Filtering

Convolution is the backbone for many of the image processing algorithms we employ. Convolution is integral for blurring the image, detecting edges, and the CNN that classifies the type of shape. In computer vision, convolution is the result of essentially sliding a kernel (which often represents a specific filter) over an image and computing the dot product of the overlapping area of the image with the kernel..

The convolution operation in computer vision is based on the mathematically rigorous definition used on continuous signals but adapted for images with discrete pixels. Specifically, the convolution of an input image I(x, y) with a kernel K(x, y) is defined as:

$$(I * K)(x, y) = \sum\sum I(i, j)\, K(x-i, y-j) \qquad (1)$$

where * denotes the convolution operator, and $\sum\sum$ denotes summation over all possible values of i and j that overlap with the kernel K at the point (x, y).

The main intuition behind convolution in computer vision is that the kernel K acts as a local filter that extracts specific features or patterns from the image. Kernels can be used to search for specific patterns (for edge detection and CNNs) or to incorporate information about surrounding pixels (for blurring, dilation, and erosion).

There are multiple kernels we used in our project. To smooth the images and reduce noise, we used a gaussian kernel. One main advantage of this kernel is that its circular symmetry makes it less computationally intensive because it can be implemented as two single-dimension convolutions instead of one expensive 2-dimensional convolutions [18]. We used a OpenMV's default 5x5 Gaussian kernel [19] which is shown in Table III.

TABLE III
GAUSSIAN KERNEL

Gaussian Kernel				
1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

The Canny edge detection algorithm, which we used to detect edges, uses the Sobel operator [20]. The Sobel operator consists of two 3x3 kernels, one for detecting edges in each the horizontal and vertical directions [21]. The idea behind this algorithm is to apply a kernel to the image that emphasizes areas where there are rapid changes in the intensity or color of neighboring pixels. Such rapid changes are typically indicative of edges or boundaries between different regions in the image. Since there are two directions in a 2-dimensional image, we need two kernels to identify rapid changes in both of these directions. The kernels are shown in Table IV.

TABLE IV
SOBEL FILTERS

Horizontal Sobel (Gx)				**Vertical Sobel (Gy)**		
-1	0	1		1	2	1
-2	0	2		0	0	0
-1	0	1		-1	-2	-1

The final edge-detected image is the magnitude of the gradient at each pixel location, which is computed as the square root of the sum of the squared horizontal and vertical gradients. This results in an image where the pixels corresponding to edges or boundaries have high values, while the rest of the image has low values.

Dilation and erosion are additional operations that use convolution. Dilation expands the boundaries of the objects in an image while erosion shrinks them. Dilation and erosion are often used together, sometimes repeatedly, as dilation can be used to fill gaps and holes in an image while erosion can be used to remove small objects or noise [22]. This is another way of smoothing out noise and inconsistencies like gaussian blurring.

We also use convolution in a convolutional neural network (CNN) which is a common tool used in deep learning for image classification tasks. In a CNN, convolutional layers apply a series of learned kernels to an input image to extract increasingly complex features. Additionally, max pooling layers downsample the feature maps to reduce the dimensionality of the input to subsequent layers. The output of the final convolutional layer is then fed into a fully connected layer that performs the final classification [23]. When designing a CNN, the architect defines the size and number of the kernels to apply to the image, and the training process learns the optimal values for the kernels.

Image Representation and Color Spaces

Representing an image in different formats allows images to be stored more compactly and simplify the process of extracting relevant information. Our system uses grayscale images, binary images produced by thresholding, and color images in the LAB color space.

Grayscale images condense the multi-channel color values into a single value describing how dark or light a pixel is. By reducing the number of color channels from 3 to 1, converting an image to grayscale reduces the memory required to store the image and the amount of data required to process the image.

Taking it one step further, grayscale images can be thresholded such that each pixel is described by a single bit. Given a threshold value, every pixel in the image is compared to that threshold and replaced with a “1” if it is greater than that threshold or “0” if it is below. Thresholded images are very helpful for running shape detection algorithms since the image is represented in a very simple, binary format.

However, hard-coded threshold values are not robust against variations in lighting conditions and color intensities, as different threshold values are required to properly threshold an image under different conditions.

Otsu’s method is a common technique for computing the ideal threshold value [24]. It works by testing out each possible threshold value and measuring the variance among the pixels above the threshold and the variance among the pixels below the threshold. The optimal threshold value is the one which minimizes the variances within the two classes [24]. For example, when determining a threshold to separate a dark blob from a light background, the variance between the two classes will be minimized when one class only contains dark pixels and the other class only light pixels.

We used Otsu’s method to determine the optimal binary threshold to isolate the colored pixels on each card from the white card background. We only need to consider the optimal threshold value for the L channel, as white will always have A and B values close to zero.

To represent color images, we use LAB color space, where "LAB" stands for the three dimensions of the color space which are “lightness”, “A” and “B” [25]. The LAB color space is designed to be perceptually uniform, meaning that distances between colors in the LAB space correspond to how humans perceive color. This makes it a useful color space for applications including color correction, color grading, and image analysis [25].

The lightness dimension (L) in the LAB color space represents the brightness of a pixel, with values ranging from 0 (black) to 100 (white). The A and B dimensions represent the green-red and blue-yellow color spectra, respectively. The A dimension ranges from -128 (green) to 128 (red), while the B dimension ranges from -128 (blue) to 128 (yellow) [24]. The A dimension is especially helpful for us because two of the possible card colors are red and green. The lightness dimension is also helpful for identifying the fill of the image since solid shapes are darkest and empty shapes are lightest.

Perspective Warping

Perspective warping, also known as perspective transform, is the process of altering the perspective of the image through interpreting an image’s 2-dimensional structure in a 3-dimensional (3D) space and then mapping relevant 3D coordinates to new 3D coordinates which are shown in a new 2-dimensional image. This is useful for applications such as correcting distortions caused by a camera lens or when creating panoramic images [26].

The perspective warping transformation is typically defined by a 3x3 matrix called the homography matrix which maps the original image points to their new positions. This matrix can be computed using the correspondences between points in the original image and their desired corresponding points in the new image [26]. We use perspective warping to take the outline of each card from the original image and transform them to a standard size of 60x90 pixels that appears as a head-on view from directly above the card.

E. Hardware

On the hardware side, we had two main components: an OpenMV H7 board and an LED matrix. The OpenMV board combines an STM32H743VI microcontroller with an OV7725 camera [27]. The camera has a maximum resolution of 640x480 pixels with 16 bits per pixel, but the resolutions and frame rate can be altered to adjust performance. The microcontroller runs at 480 MHz with 10 available GPIOs on the board. The OpenMV board also includes a micro-SD card holder and onboard RGB LED [27].

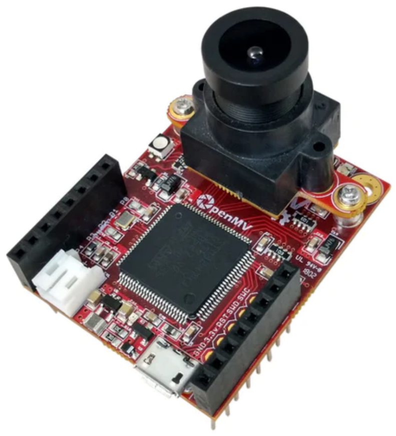

Fig. 1. OpenMV H7 board [27]

The matrix is an 8x8 green LED matrix created by Adafruit that is 1.2” x 1.2” [28]. To control this matrix through I2C, we used a matrix controller built off the HT16K33 LED driver from Adafruit [29]. Although Adafruit’s library (https://github.com/adafruit/Adafruit_CircuitPython_HT16K33) for this controller is written for CircuitPython and is not directly compatible with OpenMV, we were able to recreate the controller for the OpenMV environment without significant trouble.

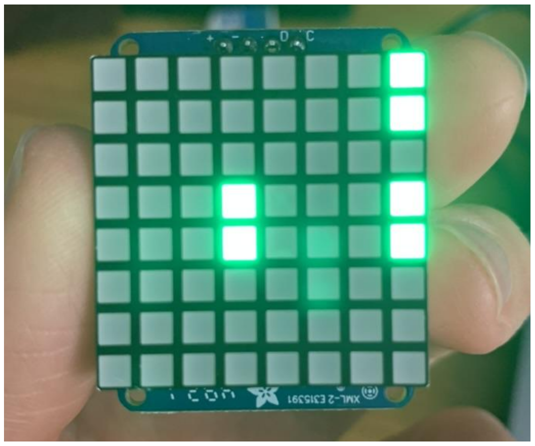

Fig. 2. 8x8 LED matrix

These two components communicate using I2C which only requires 4 wires (+3.3V, GND, SCL, SDA). Since Adafruit’s controller board has on-board pull-up resistors on the I2C data and clock lines, we didn’t need any external passives.

To power our system, we used a 3.7V lithium polymer (LiPo) battery. This plugs directly into the OpenMV board through a JST connector. To make it simpler to turn on and off the board, we added a small power switch between the battery and the board.

F. Software

On the software side, we used Google Colab to develop our algorithms and the OpenMV IDE to interface with the OpenMV board. The benefits of using Google Colab to develop the algorithms is that all of our code is automatically shared between us and that all required Python packages were already installed. Using Python in a Jupyter Notebook on Google Colab made it very fast to develop and test algorithms before implementing it in MicroPython to run on the H7.

OpenMV’s IDE is custom built to interface with the OpenMV board. It uses a custom version of MicroPython with extra libraries to support the OpenMV board. Most notably, it includes a range of image processing functions that we could use in our system. However, the functions available are not a one-to-one match for the functions available in OpenCV, so we had to substitute and adapt our algorithms when we moved our algorithms from Google Colab to MicroPython. However, there are also algorithms available in OpenMV that are not available in OpenCV which we were able to utilize to increase the robustness of our algorithms.

G. Operation

There are many steps in our software system which are briefly outlined in Fig. 3. The algorithm starts with preprocessing to aid in identifying the 12 cards. Once it finds all 12 cards, it runs the classification algorithms for each of the four attributes: number of shapes, type of shape, color, and shape fill. Once it knows the attributes of each card, it finds the sets and illuminates the LED matrix with the sets it finds.

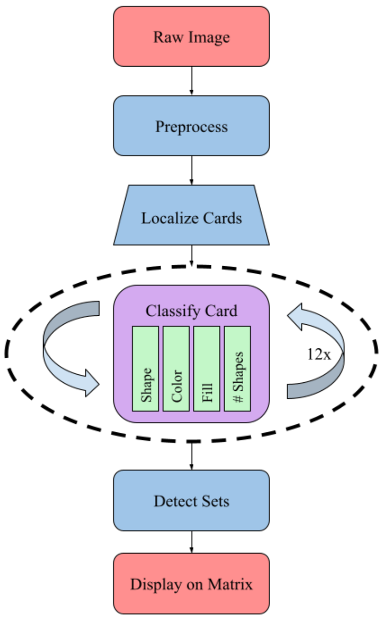

Fig. 3. System architecture block diagram

The first step in our classification algorithm is to preprocess the image to decrease noise, decreasing image size, and making relevant features clearer. Our preprocessing steps are to convert the image to grayscale, blur it, threshold the pixels, and then dilate and erode the binary pixels. The result is a binary image where the only white pixels are ones that show the edges of cards and shapes on the cards. A binary image is much smaller than a colored image because it only requires one bit to describe each pixel instead of 16 bits to describe a colored value. The OpenMV image library provides several functions that reduce these steps into a few lines of code.

```
img_gray.to_grayscale()
img_gray.gaussian(2, threshold=True, \
   offset=6)
img_gray.dilate(1)
img_gray.erode(1)
```

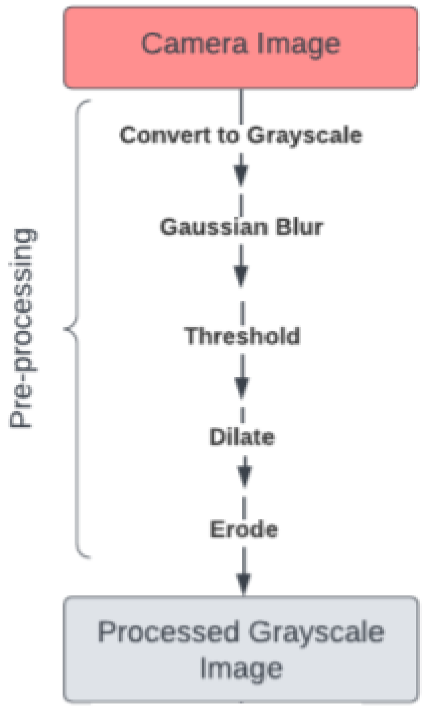

Fig. 4. Preprocessing steps

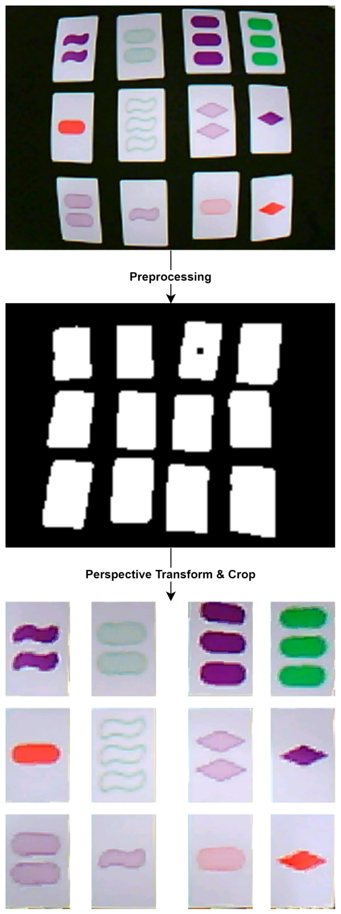

Fig. 5. Example card localization

Once it has the thresholded image, the code localizes the individual cards using OpenMV's findBlobs function which identifies contiguous regions of pixels of a certain value. We were able to filter for blobs of a certain size, ignoring any erroneous background objects that made it through thresholding. Once all the card-sized blobs had been isolated, we used OpenMV's findRects function to identify coordinates of the card corners, which we could then use to perspective warp and resize each card to be a 60x90, head-on image.

To count the number of shapes, it first runs the Canny edge detection algorithm to identify the outline of the shapes. Then, it applies various binary masks to the image and counts how many colored pixels fall outside that mask. The intuition behind this is that we know where each shape should be on the card, and we can test to see if the placement of edges align best with 1, 2, or 3 shapes.

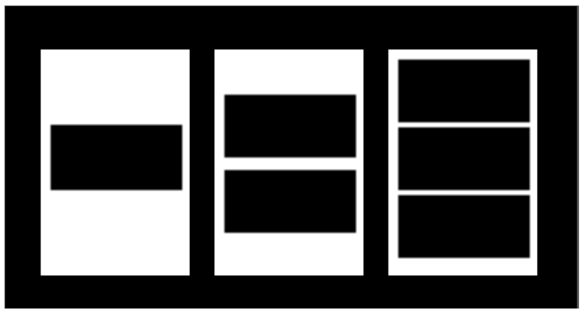

Fig. 6. Binary masks for 1, 2, and 3 shapes

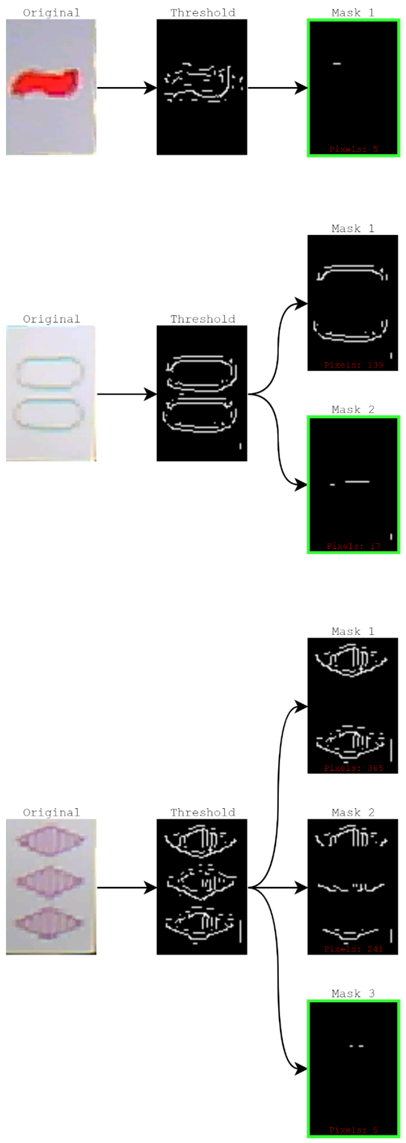

Fig. 7. Application of binary masks for images of cards with 1, 2, and 3 shapes

Once we know the number of shapes, we can extract a 24x52 pixel rectangle where each shape is. We initially tried to identify bounding rectangles using image processing libraries, but we found that we achieve the best results when we use standard shape locations based on where the shapes typically are on a card. Although this may not lead to perfectly bounded shapes, it is more robust to noise. We run this 24x52 grayscale rectangle through a CNN which classifies the shape as either an oval, diamond, or squiggle. If there are multiple shapes on the card, we run each one through the CNN and combine the results to get the end classification.

We found that the best CNN architecture for us is 2 convolution and pooling layers followed by a fully connected layer. We initially had only one convolution and pooling layer, but adding another set of convolution and pooling decreased the number of parameters due to fewer nodes entering the final fully-connected layer. Through trial and error, we found that this final architecture provided the ideal balance of classification accuracy and model complexity. Since memory and computational resources are limited on the H7, reducing complexity was a priority for us.

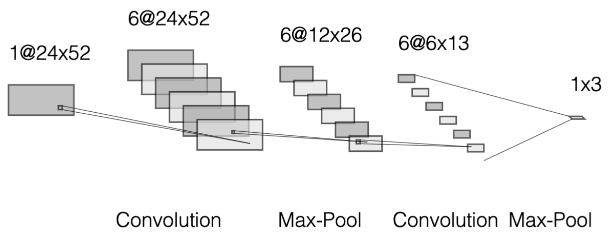

Fig. 8. Shape classification CNN architecture. Diagram generated using https://alexlenail.me/NN-SVG/LeNet.html

This CNN utilizes tensorflow lite which is a tensorflow distribution designed specifically for embedded systems. We uploaded images taken by the OpenMV board to Google Colab, augmented the dataset by cropping and rotating the images to yield 4000 total images from the original 250 training images we captured. We trained the CNN on 3200 of these images while withholding 800 of them as a validation dataset. After training for 50 epochs, we achieved 100% classification accuracy on both the training and validation dataset. While this may seem like overfitting, we are confident that the model does not overfit the data because the classification task is fairly simple and the model retained 100% accuracy on the validation dataset which it had not seen before.

Once the model was trained, we exported it as a ".tflite" file which is a binary file that can run directly on the H7. In this process we had to quantize this model so it uses 8-bit integers instead of floating point numbers. The OpenMV IDE is set up to directly run .tflite files through a single function call to the tensorflowlite library.

```
cnn_out = tf.classify(model_file, shape)
```

In this function call, "model_file" is a string that is the path to the .tflite file on the OpenMV board's filesystem, and "shape" is an "Image" object with the image to classify. This Image object is part of the OpenMV standard library and is the default type for all images captured by the onboard camera. The image we feed into the CNN has already been converted to grayscale as well as reshaped and cropped to 52x24 pixels.

For color classification, the first step is to isolate the colored pixels on each card from the background. We accomplish this using adaptive thresholding to find ideal threshold values to separate the white background from the colored shape. Next we calculate the average LAB values of the colored pixels and use the results to determine color. Green cards have negative A channel values, while red and purple cards have positive A values, so distinguishing green from red and purple is trivial. Red and purple cards have slightly overlapping B channel values depending on lighting conditions, with red tending to have slightly higher B values than purple. We determined cutoff values for red and purple through trial and error which are shown in Table V.

TABLE V
COLOR THRESHOLDS

Color	Condition
Green	A < 4
Purple	B ≤ -12, -12 < B < -4 and L < 57 - B
Red	Otherwise

Due to visual aliasing, the color in the center of striped shapes blurs into a lighter version of the solid color. We took advantage of this blurring phenomenon in our final algorithm, which simply considers the LAB values within a small 12x6 region within the center of each shape. The region is sized such that it does not include the shape boundary for any of the 3 shapes. Different decision thresholds based on the average LAB values were empirically determined for each card color.

TABLE VI
FILL THRESHOLDS

Color	Solid	Empty	Striped
Red	A > 40	A < 10, B < 5	Otherwise
Green	A < -30	A > 0, B < -2	Otherwise
Purple	A > 20	L > 85, A < 7	Otherwise

The combined attribute classification diagram is below in Fig. 9.

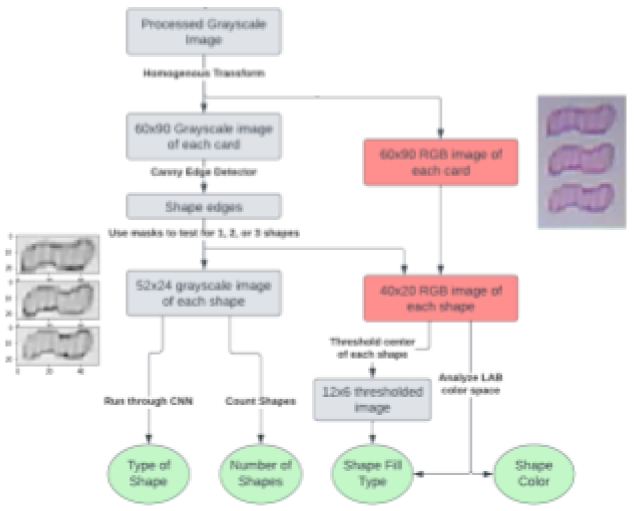
Processed Grayscale Image
Homogenous Transform
60x90 Grayscale image of each card
60x90 RGB image of each card
Canny Edge Detector
Shape edges
Use masks to test for 1, 2, or 3 shapes
52x24 grayscale image of each shape
40x20 RGB image of each shape
Threshold center of each shape
Analyze LAB color space
12x6 thresholded image
Run through CNN
Count Shapes
Type of Shape
Number of Shapes
Shape Fill Type
Shape Color

Fig. 9. Attribute classification steps

Once we know all four attributes of each of the 12 cards, we can run the set classification algorithm. Our algorithm takes advantage of two important observations: 1) given any two cards, there is only one card that will complete the set, and 2) if each variation of a card's attribute (ex: red, green, or purple) is assigned a number, then a set where each card differs in that attribute will have a consistent sum of the numerical values of that attribute. i.e. (red = 0) + (green = 1) + (purple = 2) = 3.

The set finding algorithm iterates through each pair of cards, determines which card would complete the set, and then searches the remaining unchecked cards for the desired card. Manually checking each combination of 3 cards requires iterating through all possible combinations among the 12 cards in play for a total of 220 comparisons.

Then, we display the final sets on an 8x8 green LED matrix. The code creates a Python object that controls the matrix and contains methods to set individual pixel values or patterns on the matrix. This makes it very straightforward to interact with the matrix. For example, the following 2 lines of code creates the matrix controller object and then illuminates two pixels corresponding to card “i”.

```
m = MatrixController()
m.set_card(i, True)
```

We also wrote a few functions to illuminate custom shapes. We wrote a function that converts a 2D array representing the desired matrix values into a compact 8-byte representation that can be succinctly stored on the H7. Then the H7 can read those 8 bytes and illuminate the corresponding pixels in the matrix. These custom shapes are used to create a custom startup animation that displays the word “SET” one letter at a time.

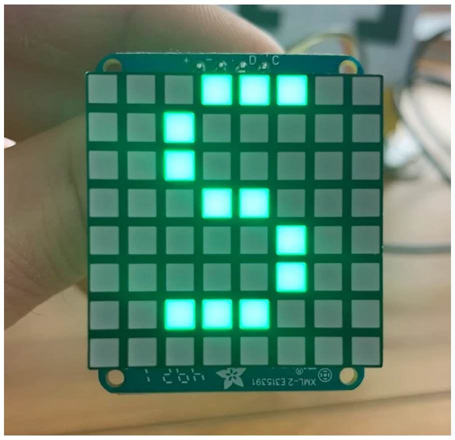

Fig. 10. The letter “S” created using out custom animation generation code

The system blinks each valid set three times so the user has time to correlate the pixels on the matrix to cards in play. If there are multiple sets, it cycles through all of them.

Links to our full code as well as a demonstration are in Appendix B.

IV. RESULTS

A. Description

We measured classification accuracy by classifying each of the 81 cards one time. The results are summarized in Table VII.

TABLE VII
CLASSIFICATION ACCURACY

CLASSIFICATION	ACCURACY
COUNT	98.8%
FILL	85.7%
COLOR	98.8%
SHAPE	100%

We defined classification speed to be the length of time between the system beginning classification and beginning to output results to the matrix. The device could classify all the cards and start displaying sets in under 5 seconds.

B. Discussion

All classifications besides fill were approximately 100% accurate. However, since all attributes of a card are important in determining whether a group of three is a valid set, the overall accuracy of the system is only as accurate as the worst performing classification, in this case fill with 85.7% accuracy. Determining whether a card has a solid fill was easy, and all incorrect fill classifications happened on empty or striped cards. Our initial approach considered the ratio of white to colored pixels in the center of the shapes. However, this technique is inconsistent because visual aliasing would sometimes cause the striped pattern to blur into a lighter version of the color. This varies across lighting conditions, as well as with how in focus the card is, which could change depending on where within the playing area a card is situated or how far the camera is held from the cards. Even after switching to color based fill classification, however, there seemed to be substantial overlap between LAB values between empty and striped cards depending on lighting conditions, making distinguishing between the two challenging.

Counting the number of shapes took several iterations to get working properly. Our first attempt involved using findBlobs to count the number of distinct blobs on the thresholded card. However, discontinuities were sometimes present in the shape boundaries that caused more blobs to be detected than shapes, especially with empty shapes. Using the shape masks proved to be much more robust, but the thresholding was sensitive to glare. Our final algorithm uses edge detection instead of color based thresholding in combination with the shape masks, and is close to 100% accurate.

Classifying the type of shape also works extremely well. It was difficult to get the CNN setup, mainly due to the need to quantize the model and export it in a format that can run on the OpenMV board. We also noticed that augmenting the training dataset with cropping and rotation significantly improved the accuracy through making the model more robust to variations in the input images. It is hard to debug a CNN when it doesn't work properly because it is near impossible to understand what happens inside, but we are very satisfied with the end performance.

Considering all processing and memory constraints, 5 seconds to completely localize and classify all cards and find sets seems reasonable. Aside from shape classification, our algorithms are not very sophisticated, as they simply consider average color values or count pixels falling outside of certain regions following thresholding and edge filtering. So, it is unlikely alternate approaches would substantially decrease classification time.

V. APPENDIX A

TABLE VIII
WEEKLY GOALS

Week	Team Goal	Tyler's Personal Goal	Bradley's Personal Goal
1	Setup development environment	Get familiar with OpenMV-H7 R1 camera module	Begin writing a classifier that can take an image of a card and identify shapes and patterns
2	Run first iteration of respective code assignments	Isolate a single card out from the background	Have a CNN on Google Colab that can successfully classify the shape present on a given card. A stretch goal is to also identify the number of shapes on that card.
3	Work on pre-processing so we have a standardized way of reducing noise in the cards.	Run card localization in realtime on the OpenMV board to determine FPS performance.	Be able to classify the shape, number of shapes, and fill of a card.
4	Get the combined classification system functioning on the H7	Enable localization of multiple cards by using SD card to store images, and implement Bradley's classification code on the OpenMV board using OpenMV image library functions	Re-implement any functions that are not automatically included by default on MicroPython.
5	Implement algorithms for whole design	Implement Bradley's classification code on the OpenMV board using OpenMV image library functions	Develop an algorithm to detect sets
6	Finish implementing classification functions in OpenMV to have a	Continue experimenting with preprocessing steps to isolate shape	Increase the accuracy given new data from the integrated system.

	unified software system	bounding boxes	
7	Determine which algorithms/approaches we want to use in the end preprocessing and classification system	Implement color and fill identification	Continue porting my classification code to OpenMV. Given additional training data from Tyler, test classification algorithms for robustness.
8	Begin working on the physical design of the end product	Improve fill classification accuracy and begin CAD for device enclosure	Find out how to run the H7 without being connected to a computer. This means finding a battery and making our program run by default.
9	Assemble physical product and improve performance through testing.	3D print enclosure and assemble system.	Begin drafting final report
10	Prepare and practice final presentation	Finish assembly and integration of set-finding + displaying code with classification code, record demo, and practice presentation.	Finish a draft of the final report

VI. Appendix B

The code we created for this project can be found on Box at: https://ucla.box.com/s/5kbnkuux7agr61hy93tcf1idzzip6ldq

A demonstration video can is located in the following folder: https://ucla.box.com/s/b68szhstse4bejrl94q6z3b9k68oqslg

References

[1] Huang, T.S. "Computer Vision: Evolution and Promise". University of Illinois at Urbana-Champaign. 1996. Retrieved from https://cds.cern.ch/record/400313/files/p21.pdf

[2] Grover, Prince. "Evolution of Object Detection and Localization Algorithms". Towards Data Science. Feb. 14 2018. Retrieved from https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad

[3] K. Kyunghwan. "Multiple Object Detection Algorithms". Institute for Advanced Architecture of Catalina. Jul. 7, 2021. Retrieve from: https://www.iaacblog.com/programs/object-detection-algorithm/

[4] S. Zickler and M. M. Veloso, "Detection and Localization of Multiple Objects," 2006 6th IEEE-RAS International Conference on Humanoid Robots, Genova, Italy, 2006, pp. 20-25, doi: 10.1109/ICHR.2006.321358.

[5] Ji, Q., Dai, C., Hou, C. et al. "Real-time embedded object detection and tracking system in Zynq SoC". J Image Video Proc. 2021. doi: 10.1186/s13640-021-00561-7

[6] Y. Sun, C. Wang and L. Qu, "An Object Detection Network for Embedded System," 2019 IEEE International Conferences on Ubiquitous Computing & Communications (IUCC) and Data Science and Computational Intelligence (DSCI) and Smart Computing, Networking and Services (SmartCNS), Shenyang, China, 2019, pp. 506-512, doi: 10.1109/IUCC/DSCI/SmartCNS.2019.00110.

[7] S. Jagannathan et al., "Efficient object detection and classification on low power embedded systems," 2017 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 2017, pp. 233-234, doi: 10.1109/ICCE.2017.7889296.

[8] B. Lent and D. MacLagan. "The Card Game SET". The Mathematical Intelligencer. 25, pp. 33-40. 2003. doi: 10.1007/BF02984846

[9] A. Jain. "CS231A Final Project Report: Set AI". Stanford University. 2021. Retrieved from https://web.stanford.edu/class/cs231a/prev_projects_2021/CS231A_Project_Progress_Report__1_.pdf

[10] C. Simler. "Building a SET Solver Using Python and OpenCV". Medium. Sep. 9 2020. Retrieved from https://medium.com/swlh/building-a-set-solver-using-python-and-opencv-4413389e7fdd

[11] N. Hahn. "SET Card Game Solver with OpenCV and Python". Nicholas Hahn. Jul. 25 2018. Retrieved from https://www.nicolas-hahn.com/recurse/center/2018/07/25/set-solver/

[12] G. Thiruvathukal and Y. Lu, "Efficient Computer Vision for Embedded Systems" in Computer, vol. 55, no. 04, pp. 15-19, 2022. doi: 10.1109/MC.2022.3145677

[13] D. Mwiti. "Top Tools to Run a Computer Vision Project". Neptune.ai. Jan. 26, 2023. Retrieved from https://neptune.ai/blog/top-tools-to-run-a-computer-vision-project

[14] P. Ingle. "Top Computer Vision Tools/Platforms in 2022". Marktechpost. Sep. 8, 2022. Retrieved from https://www.marktechpost.com/2022/09/08/top-computer-vision-tools-platforms-in-2022/

[15] G. Boesch. "The 12 Most Popular Computer Vision Tools in 2023". Visio.ai. 2023. Retrieved from https://viso.ai/computer-vision/the-most-popular-computer-vision-tools/

[16] T. DiCola. "MicroPython Basics: What is MicroPython". Adafruit Industries. Retrieved from https://www.digikey.com/en/maker/projects/micropython-basics-what-is-micropython/1f60afd88e6b44c0beb0784063f664fc

[17] J.M. Irazabal and S. Blozis. "AN102016-01. I^{2}C Manual". *Philips Semiconductors*. Mar. 24, 2003. Retrieved from https://www.nxp.com/docs/en/application-note/AN10216.pdf

[18] R. Fisher, et al. "Gaussian Smoothing". Hypermedia Image Processing Reference (HIPR). University of Edinburgh. 2003. Retrieved from https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

[19] I. Abdelkader et al. "imlib.c" OpenMV. 2022. Retrieved from https://github.com/openmv/openmv/blob/master/src/omv/imlib/imlib.c

[20] "Canny Edge Detection". OpenCV Docs. https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html

[21] R. Fisher, et al. "Sobel Edge Detector". Hypermedia Image Processing Reference (HIPR). University of Edinburgh. 2003. Retrieved from https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

[22] R. Fisher, et al. "Mathematical Morphology". Hypermedia Image Processing Reference (HIPR). University of Edinburgh. 2003. Retrieved from https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/OWENS/LECT3/node3.html

[23] "Convolutional Neural Networks (CNNs / ConvNets)". CS321n Convolutional Neural Networks for Visual Recognition. Stanford University. Retrieved from https://cs231n.github.io/convolutional-networks/

[24] N. Otsu, "A Threshold Selection Method from Gray-Level Histograms," in IEEE Transactions on Systems, Man, and Cybernetics, vol. 9, no. 1, pp. 62-66, Jan. 1979, doi: 10.1109/TSMC.1979.4310076.

[25] A. Martin. "4.4 Lab Colour Space and Delta E Measurements". BC Campus. Retrieved from https://opentextbc.ca/graphicdesign/chapter/4-4-lab-colour-space-and-delta-e-measurements/

[26] S. Soatto and A. Kadambi. "Homography, Perspective, RANSAC". CS 188 Introduction to Computer Vision, UCLA. 2021

[27] "OpenMV Cam H7". OpenMV. 2023. Retrieved from https://openmv.io/products/openmv-cam-h7

[28] "1.2" 8x8 Matrix Square Pixel - Pure Green - KWM-R30881CPGB". Adafruit Industries. Retrieved from https://www.adafruit.com/product/1820

[29] "Adafruit 1.2" 8x8 LED Matrix Backpack". Adafruit Industries. Retrieved from https://www.adafruit.com/product/1048