CS231A Final Project Report: Set AI

Aditi Jain
ajain4@stanford.edu
Winter 2021

Abstract

SET is a card game where the goal is to match shapes and patterns to form groupings, or "Sets", of three cards which conform to a special property. Humans are extremely quick at recognizing visual similarities between the SET cards, making the game a great candidate for artificial intelligence. In this paper, we leverage the full computer vision toolkit, from low-level image processing such as edge-detection to deep learning convolutional neural networks. We observe that an AI is indeed able to play SET with the accuracy and speed comparable to that of a human.

1. Introduction

SET[1] is a card game with 81 different cards. Each card has a unique value (of three possible values) across four different attributes ($3^4 = 81$). The four attributes are **color**, **shape**, **fill**, and **number**. **color** takes on the three colors *red*, *green*, *purple*, **shape** takes on *diamond*, *oval*, *squiggle*, **fill** takes on *solid*, *striped*, *none* and finally **number** takes on *1*, *2* and *3*. Players compete on a 3 x 4 grid of SET cards to find the most Sets, where a set consists of three cards which are either all the same across each attribute, or all different. This criteria for a valid set is called the magic rule.

For example, the three cards below form a Set since they are all different along color, shape, fill, and number.

Figure 1. These three cards form a Set since they are either all different or all the same across the four dimensions (color, shape, fill, number).

As sets get removed from the game board, the spaces are backfilled with new cards from the deck. The game finishes when there are no more cards in the deck and no more Sets can be found on the board. The player with the most Sets wins.

1.1. Problem Statement

The problem at hand is to teach an AI to play SET. This entails segmenting individual cards from the twelve card game board, classifying the color, shape, fill, and number, and finally correctly forming sets across cards based on the magic rule.

1.2. Success Criteria

We evaluate the AI approach on (a) the end-to-end latency and (b) accuracy of the final Set. For reference, an experienced player can find sets in roughly 3-5 seconds, based on personal experience. In this paper we will also share the latency and accuracy of the individual steps (segmentation, classification, and set forming).

2. Background and Related Work

Image segmentation is the process of extraction objects of interest from an image. Our use case of extracting a playing card is on the simpler end of segmentation problems, as the card is always shaped as a rectangle and its white color clearly stands out against a dark background. Segmentation is well-studied and libraries such as OpenCV expose out of the box methods to extract regions of interest. In this paper, we explore both out of the box methods and segmentation implemented from scratch.

Image classification via Convolutional Neutral Networks is the process of predicting the label for a given image, given a vast amount of labeled training data. In this paper we compare transfer learning via MobileNetV2[2] with a CNN built from scratch.

Finally, let us briefly touch on two tried approaches, published as blog posts, for solving SET via artificial intelligence. In [3], the author used `cv.findContours` for segmentation. The author then attempted to predict

each individual attribute (**color**, **shape**, **fill**, **number**) with a custom technique. For example, they wrote an algorithm to distill the average color, and leveraged OpenCV's ORB[4] to detect keypoints forming the shape. The author did not comment on the accuracy of the approach.

In [5], the author used out of the box computer vision libraries to extract the cards from a black matte surface. The author then curated a training dataset of 160 images per card with various lightings and angles, and used it to train a CNN model to predict the given card. The model reached 99% accuracy. This approach is more along the lines of what you will see in this paper, but with a twist of generating training data via augmentation.

3. Technical Approach

We break down the problem into four major technical sub-problems: card segmentation, training data generation, card classification, and Set formation.

3.1. Card Segmentation

Given an arbitrary SET game board (see below), we need to extract each individual card for downstream classification.

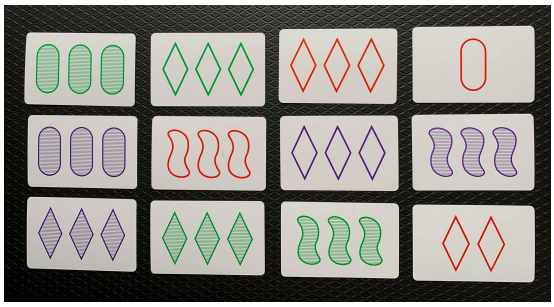

Figure 2. An example Set game board. Card extraction requires segmenting each card.

3.1.1 Segmentation via cv.findContours

In the first approach, we used the following algorithm:

1. Convert the image to grayscale.
2. Threshold the image to make the objects more distinct.
3. Run `cv.findContours`.
4. Extract the cards as the 12 largest contours by area.

We will discuss the results in a subsequent section.

3.1.2 Segmentation via from Scratch

To understand more what was going on behind the scenes of `findContours`, I implemented card extraction by scratch:

1. Convert the image to grayscale.
2. Threshold the image to make objects more distinct.
3. Run Canny Edge detection[6] (implemented by hand).
   (a) Apply smoothing to remove noise.
   (b) Calculate image gradients to highlight the edges.
   (c) Use non-maximum suppression to thin the edges.
4. Fit the edges to lines using Hough[7] transforms.
5. Filter out lines which are not horizontal or vertical.
6. Compute intersection points. These points form the corner points of the cards.
7. Extract the cards!

In the Results section, we will show the visualizations of the intermediate steps as well a comparison of the final result against `cv.findContours`.

3.2. Training Data Generation

Before training a classifier to identify each individual card as one of 81 classes, we need to generate training data. I started by taking a picture of each individual card and giving it a unique class label between 0 and 81. To generate this numerical label, I utilized the Base 3 mapping between a 4-digit string e.g. "1210" and a decimal number between 0 and 80. The 4-digit string specifies the 4 attributes and the value at index i represents the value taken on by ith attribute. See `class_to_string` in the accompanying Colab notebook.

From each card's picture and corresponding model label, I generated 300 copies of each card with random lighting and geometric transformations. While I had first experimented with random rotation and scaling in addition to the following transformations, I found that the model had a difficult time learning these variations. I was ultimately able to remove these transformations with no cost to performance during inference.

Algorithm 1: Training Data Augmentation

```
gamma = [0.5, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.5]
shears = [-0.03, -0.02, -0.01, 0, 0.01, 0.02, 0.03]
flips = [0, 1, -1]
sigmoid_cutoffs = [0.2, 0.4, 0.5, 0.6]

# Random exposure change
image = exposure.adjust_gamma(image,
random.choice(gamma))

# Random Sigmoid Correction
image = exposure.adjust_sigmoid(image,
random.choice(sigmoid_cutoffs))

# Random Flip (or no flip)
flip = random.choice(flips)
if flip ≥ 0: image = cv.flip(image, flip)

# Random Affine Transform Shear.
affine = transformÀffineTransform(shear =
random.choice(shears))
warped = transform.warp(image,  inverse_map=affine,
preserve_range=True)
```

We ran this algorithm 300 times for each of the 81 cards, yielding a training data set of 24k images.

3.3. Card Classification

To classify the SET card as one of 81 different classes, I tried the following two approaches, eventually settling on the second. Both used 80/20 split between training and validation.

3.3.1 Classification via Fine-Tuning on a Pre-Trained Model

As I was initially aiming to deploy a model on device to play SET in real-time, I leveraged MobileNetV2 [2] trained on ImageNet as the weights for the CNN, with 40% dropout and an additional layer for softmax. Below is the model summary.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 1280)              2257984
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 81)                103761
=================================================================
Total params: 2,361,745
Trainable params: 103,761
Non-trainable params: 2,257,984
_________________________________________________________________
```

Figure 3. Image classification via MobilenetV2.

3.3.2 Classification via from-scratch CNN

I compared the approach above with the convolutional neural network structure used in [5] which was pulled from Chapter 5 of Deep Learning with Python[8]. The network stucture is shown below.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 222, 222, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 111, 111, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 109, 109, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 54, 54, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 52, 52, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 26, 26, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 24, 24, 128)       147584
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 12, 12, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 18432)             0
_________________________________________________________________
dropout (Dropout)            (None, 18432)             0
_________________________________________________________________
dense (Dense)                (None, 512)               9437696
_________________________________________________________________
dense_1 (Dense)              (None, 81)                41553
=================================================================
Total params: 9,720,081
Trainable params: 9,720,081
Non-trainable params: 0
_________________________________________________________________
```

Figure 4. Image classification via From-Scratch CNN.

3.4. Set formation

The final step is straightforward; once we have the predicted class for each card on the game board, we iterate through all combinations of three cards and check if the magic rule criterion is met. As the board contains 12 cards, there are a total of 12c3 = 220 combinations to check, which is computationally tractable for our use case. I find that humans generally do something smarter than considering each combination independently; a common shortcut is to look for ”minorities” on the board - e.g. there may be only one card with two objects - and search through Sets which may contain that card. As a fun extension, the problem of finding disjoint Sets was proven to be NP-Complete by researchers at Kyoto University and CUNY[9].

4. Results

Next, we will detail the results achieved by each of the technical subparts.

4.1. Card Segmentation

4.1.1 Segmentation via cv.findContours

The results of segmentation via cv.findContours is shown below. Note that we we filter to the 12 largest contours by area to remove the shapes captured on the cards.

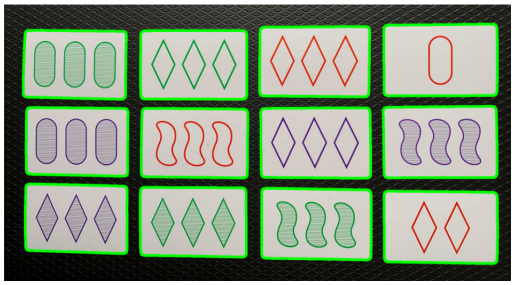

Figure 5. Segmentation via cv.findContours.

This algorithm takes roughly 1 second to execute. The code itself was very easy to implement, only requiring four lines of Python!

4.1.2 Segmentation from Scratch

The major steps to this implementation are (1) Canny Edge Detection, (2) Line fitting via Hough transforms, and (3) Computing intersection points and extracting each of the 12 cards.

1. Canny Edge Detection
   The notable parameters we used are (1) thresholding the grayscale image at 160 and (2) applying a mean filter for smoothing with 5x5 kernel size. The results of Canny Edge Detection are shown below. This piece of code took 5 seconds to execute.

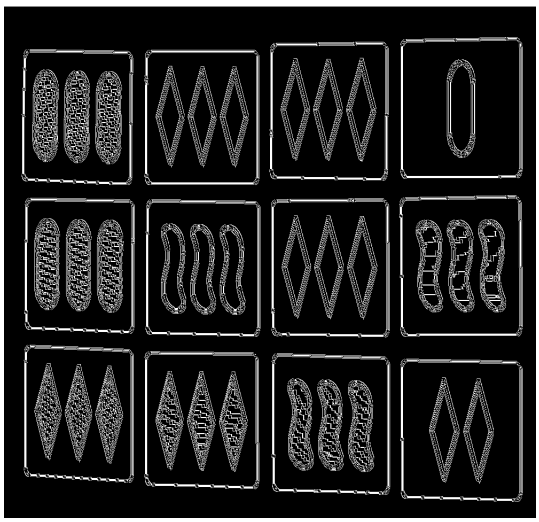

Figure 6. Board after Canny Edge Detection.

2. Line Fitting via Hough Transforms
   In the second step, we pass the black-and-white image with the extracted edges to `cv.HoughLines` to find lines parameterized by polar coordinates. We use Hough Transforms as we have vertical lines (infinite slope) in the image, which are not supported by the Cartesian Coordinate System. The parameters we pass to `cv.HoughLines` are `rho=1`, `theta=np.pi/180`, `threshold=325`. These parameters were found through trial and error experimentation.

   The call to `cv.HoughLines` returned a total of 35 lines. After post-processing to remove lines which were not nearly horizontal or nearly vertical (± 15 degrees), we ended up with the following straight lines.

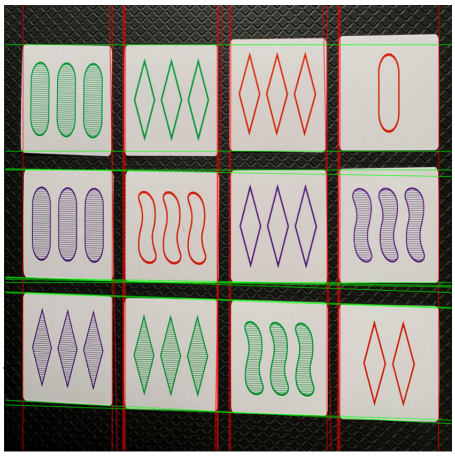

Figure 7. Straight Lines extracted via cv.HoughLines.

We then computed the intersection points between each pair of horizontal and vertical Hough lines. These points are visualized in yellow.

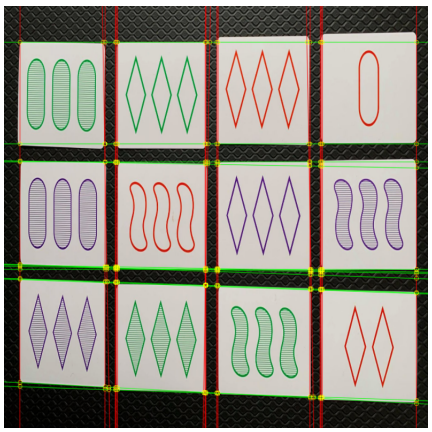

Figure 8. Intersection points (yellow) computed between vertical and horizontal Hough Lines.

We then run a suppression step to merge intersection points which are close together into a single point, forming the corner of the card. See the merged points,

in aqua, below. Note that there are 48, as desired (12 cards x 4 corners).

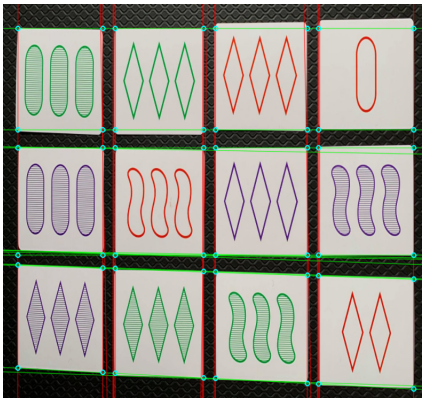

Figure 9. Merged intersection points (aqua) computed between vertical and horizontal Hough Lines.

3. Card Extraction
   Finally, we need to group each of the 48 points to their parent SET card. We use a simple algorithm that attempts to extract each horizontal row of points at a time (of six rows), and assign each successive pairing of two points to the same card. After this step, we end up with the following 12 contours!

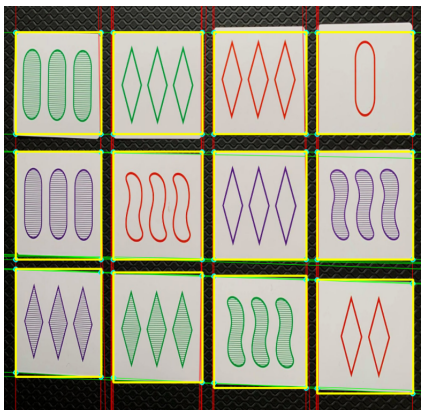

Figure 10. Cards extracted (yellow) via Hough Transformations.

The total time to run Hough Line fitting and Card Extraction is 3 seconds, plus the additional 5 seconds for Canny Edge Detection. Comparing this to `cv.findContours`, we see that my implementation is much slower (8 seconds compared to 1 second). I would attribute this difference to inefficient calculations (e.g. for-looping through images rather than optimized matrix operations). Additionally, looking at the output of `cv.findContours` vs my Hough Lines approach, we also notice that the former finds the exact contours of the card, while the latter chops up the edges of the card in some cases. I could have gotten around this by requiring a smaller number of points for fitting a Hough line, but at the cost of picking up many other lines in the image shapes contained within the cards. I find that the non-exact crops does not affect the performance of the downstream classifier.

4.2. Card Classification

As described above, we tried two models: MobileNetV2 pre-trained on ImageNet and a from-scratch CNN. Both models used an 80/20 split and 300 images per class (of 81 classes). The former required 224 x 224 image size while the latter required resizing to 150 x 150 pixels. Both models were trained for 10 epochs and with 128 batch size.

The Test set is the set of 12 cards extracted from a single SET game board, specifically the one displayed in preceding sections.

Table 1. Model Accuracy of Card Classification

	Training	Validation	Test
MobileNetV2	99%	99%	42%
From-Scratch CNN	99%	99%	92%

While both models seemed to have performed very well on the augmented training and validation sets, MobileNetV2 performed significantly worse on new data, predicting only 5 of 12 cards correctly.

For a deeper comparison, here is the per-card prediction the models applied to each card in the test set.

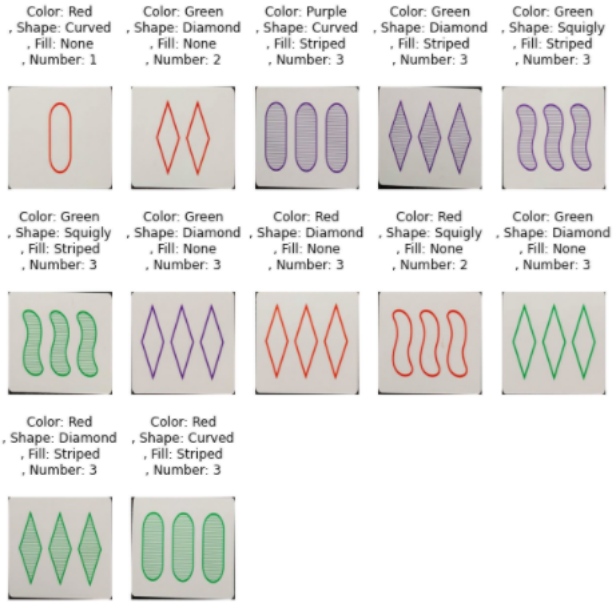

Figure 11. Predictions on Test Set from MobileNetV2.

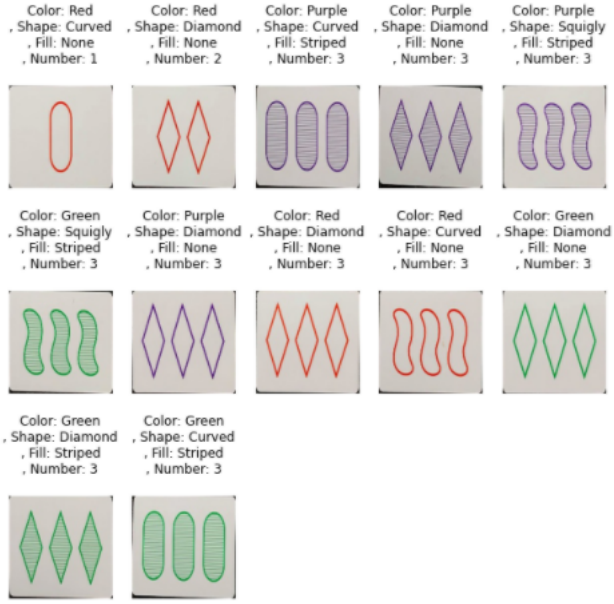

Figure 12. Predictions on Test Set from From-Scratch CNN.

My guess is that MobileNetV2 was merely memorizing the augmented training data set. For the final model, we ended up choosing the From-Scratch CNN. Regarding inference latency, MobileNetV2 took 3 seconds to run inference on 12 images, whereas From-Scratch took 2 seconds.

4.3. Testing with Various Game Boards

Finally, we run the end-to-end flow (Segmentation → Classification → Set Forming) through 4 game boards with various cards, lightings, and background. For segmentation we used the out of the box `cv.findContours` and for classification, the From-Scratch CNN.

The metrics are shown below:

Table 2. End-to-End Metrics

	Set Found?	Was the Set correct?	Latency
Board 1	Yes	Yes	1.9 seconds
Board 2	Yes	Yes	2.1 seconds
Board 3	Yes	Yes	2.4 seconds
Board 4	Yes	No	1.9 seconds
Board 5	Yes	Yes	1.7 seconds

Based on just a few data points, the accuracy of the approach was 80% and average latency was exactly 2 seconds.

Here are two interesting failure cases which arose while testing out the game boards:

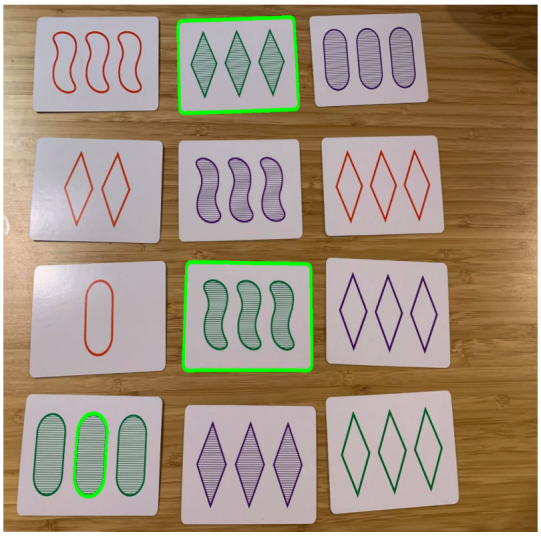

Figure 13. In this ”failure” case, the segmentation step chose an individual shape as the card, rather than the entire card. As it turns out, the model still predicted this card and formed the Set correctly!

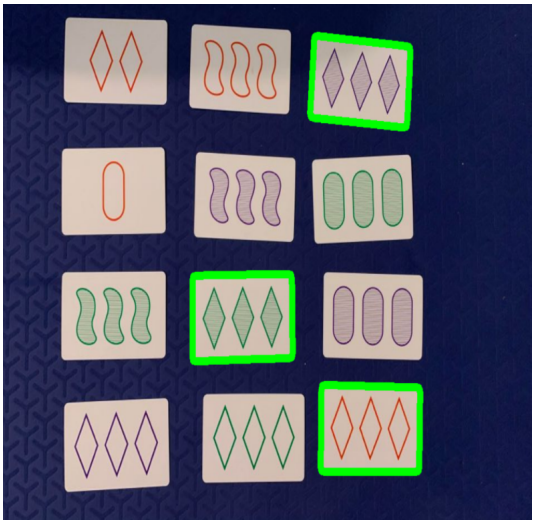

Figure 14. In this real failure case, the model predicted the bottom right card incorrectly as having stripes rather than having no fill.

An interesting extension to improve the accuracy would be to choose the Set with the highest average model confidence, rather than taking the first Set we find. In a real game, a penalty is sometimes incurred when accidentally picking up cards that do not form a valid Set, and in this type of scenario we would want the AI to be fairly confident about the Sets that it is choosing.

5. Conclusion

This paper proves the ability of an AI to play the card game SET at a level comparable to that of humans. The approach leverages out of the box image segmentation libraries in OpenCV to extract the cards from the playing surface, image data augmentation to build a robust classifier, and a standard CNN architecture for card recognition. In 4 out of 5 game boards, the approach correctly produced a valid Set, with end-to-end average latency of 2 seconds.

I learned that image segmentation by hand is a difficult problem. While extracting the edges of an image was fairly straight forward, mapping these to crisp lines requires intense post-processing to get exactly the lines we want e.g. filtering out extraneous lines which are not horizontal or vertical. It is great that out of the box approaches exist for segmentation and that they are comparable or better to my hand-tuned implementation.

The augmented training data application enabled me to go deeper into image transformations, a topic that we had touched on early on in CS231A. Generating random Affine transformations was crucial to building a robust model.

To fully prove the ability of an AI to play SET, the most crucial next step would be to deploy the algorithm to run end-to-end on device. The model would then have to keep up with the game board rapidly changing second by second as other players find and remove Sets.

At this point, we may also want to try a combined segmentation + classification + Set formation model. We could even leverage the existing models to automatically produce the training data.

6. Resources

All code was implemented and run within the Colab GPU environment. The GitHub Repository containing the notebook is available here: https://github.com/aditij113/SetAI.

References

[1] Wikipedia contributors. Set (card game) — Wikipedia, the free encyclopedia, 2021. [Online; accessed 10-Mar-2021].

[2] M. Sandler et al. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019. [Online; accessed 10-Mar-2021].

[3] Nicolas Hahn. Set card game solver with opencv and python, 2018. [Online; accessed 10-Mar-2021].

[4] OpenCV. Orb (oriented fast and rotated brief), 2014. [Online; accessed 10-Mar-2021].

[5] Tom White. Understanding how deep learning learns to play set, 2017. [Online; accessed 10-Mar-2021].

[6] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986.

[7] P. V. C. Hough. Machine Analysis of Bubble Chamber Pictures. *Conf. Proc. C*, 590914:554–558, 1959.

[8] Francois Chollet. *Deep learning with Python*. Manning, 2018.

[9] Michael Lampis and Valia Mitsou. The computational complexity of the game of set and its theoretical applications, Sep 2013.