ENPH 353 Final Report

Charting Our Path: The Evolution of Software Implementation for the Automated Robot Detective in the Engineering Physics 3rd-Year Project Course

Mahdi Shakouri & Sasan Ghasaei

University of British Columbia

Instructor: Miti Isbasescu

Fall 2023

Table of Contents

Introduction

Background

The Engineering Physics 3rd year project course is an introductory course to machine learning (ML) and neural networks (NNs). Throughout the course, students explore the different aspects of artificial intelligence (AI) by completing hands-on computer labs implementing backpropagation algorithms, convolutional neural networks (CNNs), reinforcement learning and Q-learning. Learnings through these labs build up to complete a project where students write and implement machine learning modules to automate a robot car which will drive in a simulated ROS environment and detect and guess clues to solve a crime.

Project Overview

The ENPH 353 project is a software and machine learning design project in which students in teams of two implement a software package with machine learning and image processing to compete in a simulated competition against other teams. As seen in Figure 1, the simulation environment is a route and the landscape contains a paved road, trees, active components such as a pedestrian, and a number of clue boards which the teams are tasked with reading.
In this project, we decided to detect the clue topics and clue values by cropping each letter and guessing them through a convolutional neural network.

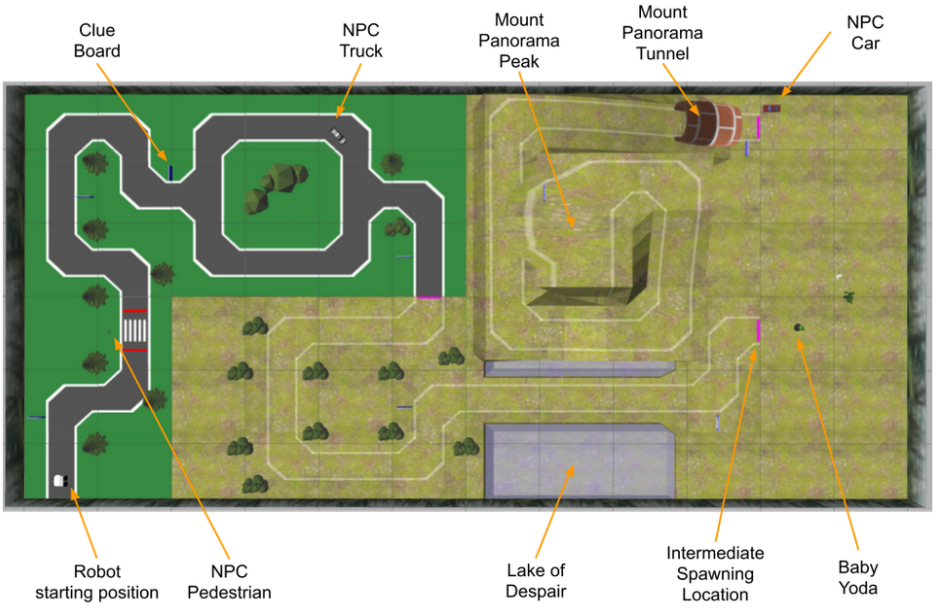


Figure 1 - The competition simulation environment

The teams are tasked to automate the movement of the robot in the environment which is equipped with a camera to navigate through the course and collect the words which are on the blue clue boards as captured in Figure 2 next page.

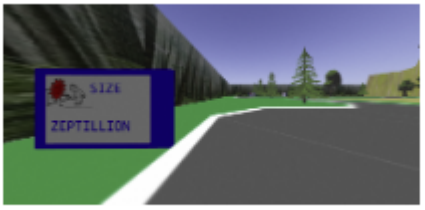


Fig 2 - An image of a clueboard as captured by robot camera

The robots are penalized for going off the track whether during the paved section or off-road of the course, and they collect points by detecting the clue boards and correctly identifying the clues, the words, on each of the 8 clue boards and submitting their findings to a clue board. The robots must also avoid hitting the pedestrian, the truck, and the baby yoda on the course.

Contributions

Mahdi's contributions to our project came in the form of the following:

- Wrote a Driver module which had three different Driver classes for each segment of the course, namely the paved, off-road, and mountain sections.
    - The module initializes a velocity publisher and all movement commands are given through this module
- Implemented computer vision algorithms to identify road segments.
    - Implemented contour analysis, morphological operations, and image noise filtering to identify the road boundaries in the simulation track.
- Wrote a Score Keeper module to communicate with the the score keeper
- Implemented a confidence thresafety net for guessing clues
- Worked on the ImageProcessor package to improve feature recognitions
- Implemented changing states in the implementation of the StateMachine

Sasan's contributions to our project came in the form of the following:

- Implemented a CNN module for character recognition of the clues.
    - Produced multiple datasets of characters for training CNN models
    - Collected data from the simulation for biases and trained multiple CNN models
- Wrote a ClueFinder module to detect and segment clueboards
    - Wrote image processing algorithms including perspective transform to detect and obtain the coordinates of the clue boards from the camera feed
- Implemented a ClueGuesser module to guess the characters of the clues
    - Wrote code to decrypt the clue images into their words using trained CNN
- Implemented a state machine for the competition controller

Software Design Architecture

Approach

One of our main goals in the design of our software architecture was to use the concept of encapsulation to make our code reusable and maintainable. We tried to promote a modular software architecture by organizing the code in self-contained units. To tackle our task of automating the robot and collecting the clue data, we split our workflow into two streams:

1. Developing and optimizing a convolution neural net model to recognize the clue values by generating and collecting a dataset suitable for the training of a CNN model.
2. Implementing a modular software for automating the robots navigation and data collection during the competition run in the simulation environment.

Overarching Software Architecture

As discussed previously, our work is separated into two main packages, each of which have their own intricacies. In the diagram in Figure 3, we describe our overarching software architecture.

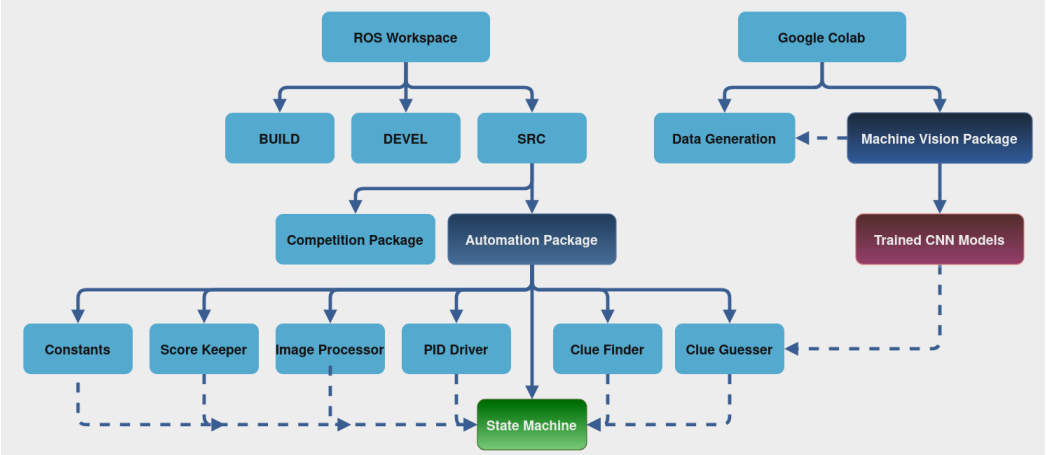


Fig 3 - Overarching Software Architecture

Machine Vision Package

Prior to discussing different software modules and structure of the code, it is worthwhile discussing our approach on training a convolutional neural network with almost 100% accuracy. We used Google Collaboratory to use a fast GPU for the training process. The CNN has nearly 800,000 parameters. The detailed structure of the Neural Network is provided in Appendix A. We performed four rounds of training on the CNN:

1. Initial Round of Training: In the first round of training, we used the empty clue banner with no letters on it, and we populated them with numbers from 0-9 and all letters of the alphabet. Since clue values could be a maximum of 12 characters, we created images with 12 characters too. An example can be seen in Figure 4 with multiple orientations.

We created similar images for other characters. After that, we also made three other copies of such images to add some blur, shift letters to the right and to the left. Below are examples of the operations done on the populated banner of A's:

Fig 4 - From left to right we have: blurry image, blurry and right-shifted image, blurry and left-shifted image

We then cropped each letter in each image and passed it to the CNN as the input along with a one-hot encoding with 37 entries indicating the letter that is being passed (Note that there are 10 digits, 26 letters and 1 space character).
Furthermore, for some letters that were similar, such as O and Q we made more copies so that the model receives more training data on characters that are similar. In total we created 48 samples for each character (since there are 4 images each with 12 characters) and for a few of the characters we made an extra copy of the image (hence 60 samples). In the first round of training, we trained for 50 epochs and with a batch size of 16. During all rounds of training the learning rate was set to 0.0001. Graphs below show the accuracy and loss of the model during training. As can be seen, the training loss starts high and decreases as training continues. The accuracy starts low and increases as training continues.

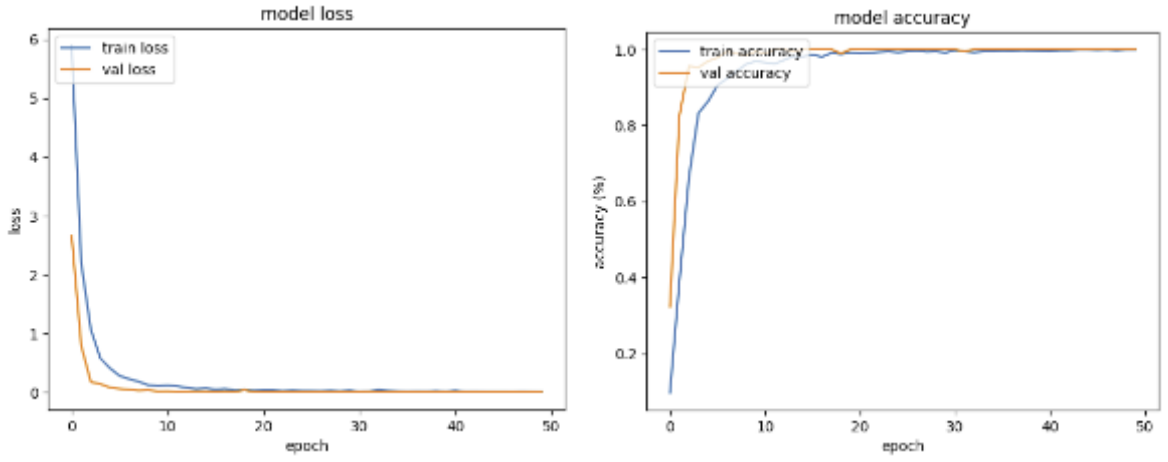


Fig 5 - Train/Val loss vs epochs on the left - Train/Val accuracy vs epochs on the right

After the first round of training, the confusion matrix for the model was perfectly diagonal indicating that the model had high accuracy in predicting images in the validation data set (to see all detailed confusion matrices refer to Appendix D). However, testing the model on images from the actual

competition did not lead to satisfactory results, prompting us that we may need to tune our model on some actual competition images too.

2. Tuning CNN on the images directly captured from the robot camera:

We then continued training the model but this time with images captured from the robot camera. We modified the clue generation code so that it would create clues with all the characters that the robot is likely to see. We created banners with three different clue values, “ABCDEFGHIJKL”, “MNOPQRSTUVWX”, “YZ0123456789” and we captured images of these banners at all 8 locations. We accumulated nearly 200 images and then we tuned the neural network with these images. Here are some examples of the images taken:

Fig 6 - some examples of images taken by the robot after clue generation code being modified

We cropped the letters of these images (only the bottom clue values) and trained the CNN on these letters. The validation accuracy of the model dropped significantly after it was trained on the new data. Moreover, the more we trained CNN on the new data, the less accurate it would become. To prevent the validation accuracy from dropping significantly, we trained the model for 15 epochs with the same batch size as before. By computing the confusion matrix after the second round of training, It turned out that the model had difficulty resolving between characters that are similar. These characters were “O,Q, 0” and “I, 1”, “S,5”, and “P,R”. This issue led us to carry out a third round of training, specifically focusing on letters that look similar.

3. Resolving Resemblance:

In this 3rd round of training, we modified the clue generation code to only generate characters that the CNN has difficulty telling apart. Similar to stage 2 of training, we drove the robot around the track and captured images of the clue banners. Below is an example of images captured to resolve the confusion of CNN between 0, O and Q:

Fig 7 - images captured to train the CNN on specific characters

In this stage of training, we had to ensure that we do not train the model for too long, since that would introduce new bias to the network and perhaps make the neural network unreliable for predicting other characters. We only captured two images for each character and we trained the model for only 5 epochs and batch size of 16. We noticed that the validation accuracy after this set of training increased significantly (above 0.95). At this point, we downloaded the model from Google Colab and used it in the competition environment.

4. Optimization:

After running the robot for multiple rounds and analyzing the performance of the CNN, we noticed that on some occasions the accuracy of the CNN drops for some specific letters of a few specific

words. We gathered all those few cases where the CNN was not perfect, and we trained the CNN only on those specific letters for many fewer epochs (between 1-3 epochs).

Automation Package

To maintain modularity, we organized the automation package into the following modules:

Constants

The Constants module is created to initialize parameters for different purposes. These parameters are instantiated with All Caps letters to indicate that they are meant to be constant and not changed throughout the structure of the code. Within Constants, four different classes are implemented to store constant values (refer to appendix C for details):

1. States: This class simply stores the different states between which the robot will need to switch throughout the course of the competition.
2. ClueConstants: This class stores constants related to the clue banner, topic and values, such as required dimensions of images, blurriness threshold, locations for cropping the letters, etc.
3. ImageConstants: This class stores constants related to filters of images in different circumstances. For example, filtering thresholds required for detecting the white lines in different parts of the track. It also contains threshold values for detecting different colors, as well as detecting the pedestrian and the truck.
4. CNNConstants: This class stores constants related to the convolutional neural network, such as array of letters that must be cropped and total number of characters to be guessed.

State Machine

Ensuring code reusability and ease of maintenance is crucial in software architecture. A state machine emerged as a key tool for well-structured code during our project. Recognizing the need for the robot to switch between different code implementations for effective navigation, we opted for a state machine. This choice enhances code modularity and reusability, offering a more maintainable and reliable system. The diagram on the next page illustrates the transitions in our robot's behavior.

Flow Chart of the State Machine Logic

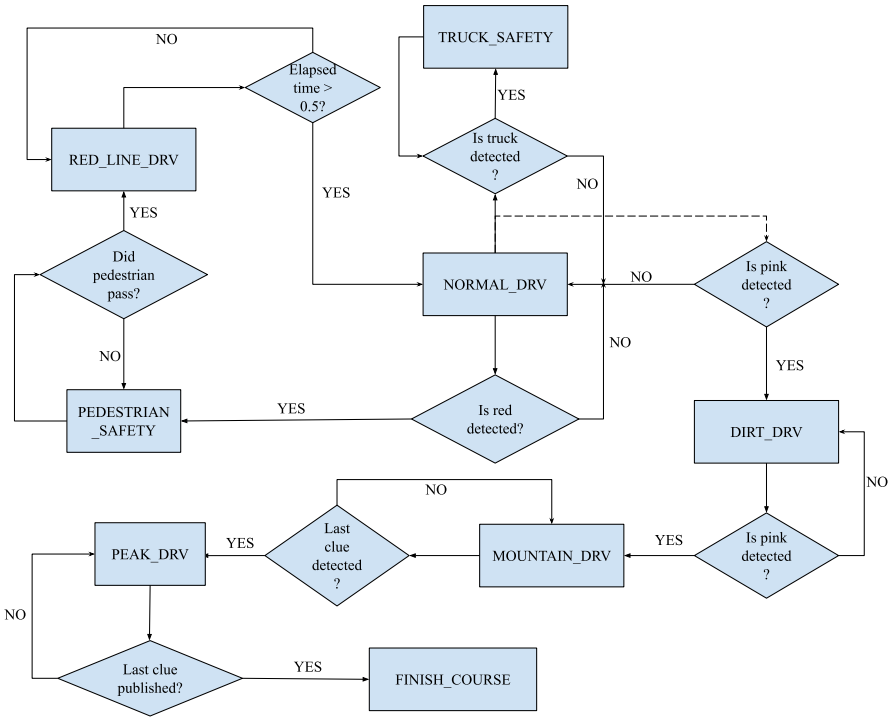


Fig 8 - State Machine Flow Chart

The state machine consists of 8 different states, as explained below:

1. NORMAL_DRV: This is the initial state that the robot starts in which involves the PID controller designed for the paved road. There are, however, some obstacles present which the robot must avoid such as the pedestrian and the truck.
2. PEDESTRIAN_SAFETY: After the first red line is detected, the robot switches to this state and waits for the pedestrian to pass.
3. RED_LINE_DRV: After the pedestrian has passed, the robot speeds up for 0.5 seconds and continues on a straight line and then switches back to NORMAL_DRV state.
4. TRUCK_SAFETY: If we detect the truck in a close distance. The robot is brought to a halt until the truck gets far away and is out of sight.
5. DIRT_DRV: If in the NORMAL_DRV state a pink line is detected, then the robot switches to driving with a PID algorithm and road thresholding specific for the dirt road.
6. MOUNTAIN_DRV: If a pink line is detected while in the DIRT_DRV state, the robot switches to the MOUNTAIN_DRV state which makes the robot to respawn to the last pink line (jumping over the Yoda) and then uses different thresholding for line detection and navigation.
7. PEAK_DRV: After the robot reaches the top of the mountain and it detects the clue banner on the top of the mountain, it switches to PEAK_DRV. In this state, we use a PID controller to drive directly towards the clue banner and get closer and closer to it.
8. FINISH_COURSE: Once a clue guess is complete and meets the criteria set by ClueGuesser, we stop the timer and the robot.

It is important to note that the process of clue guessing occurs in all of these states as the robot is driving.

Image Processor

Image Processor is a class whose functions can be used multiple times across many different classes. Here is a breakdown of some of the functions and their uses in different classes:

- For finding the banner:
  blue_filter: applies a blue mask to the input image (robot camera image)
  do_perspective_transform: given an image, the coordinates of the regions of the image that should go under transformation and the desired width and height of the output image, this function creates a transformation matrix to output an image with the right orientation.
  is_blurry: given an image, this function uses the variance of the laplacian, namely *cv2.Laplacian(gray_image, cv2.CV_64F).var()*. The idea is that sharp images have higher variance in their Laplacian compared to blurry images. A higher Laplacian variance indicates more pronounced changes in intensity, which is associated with details. See Appendix B for more details.
- Detecting the pedestrian: We use a specific thresholding to create a white background for the pedestrian and we count the number of black pixels in a specified region rectangle on the image. As soon as the number of black pixels varies, we know that the pedestrian has passed and it is safe to move on.

Fig 9 - Pedestrian being detected, the black vertical rectangle is the region that the robot is looking for variations

- Truck Detection: We use a specific thresholding for the truck, so that only the truck is visible and all other objects in the frame are filtered out. We count the number of white pixels and if it passes a threshold, we know that the truck is close and we need to stop.

Fig 10 - left picture is the truck on robot camera image and right picture is the HSV mask

ClueBoard Recognition (Clue Finder)

In the ClueFinder class, the clue banner is located and a well-cropped image of the banner with perfect orientation is returned to be passed to the Clue Guesser class to be processed and eventually guessed.

ClueFinder class does not have a constructor and its functions can be accessed directly. The most important function of this class is *"get_banner_image"*. The argument to this function is the camera image of the robot and the output is the well-cropped image of the clue banner, as show in the figure below:

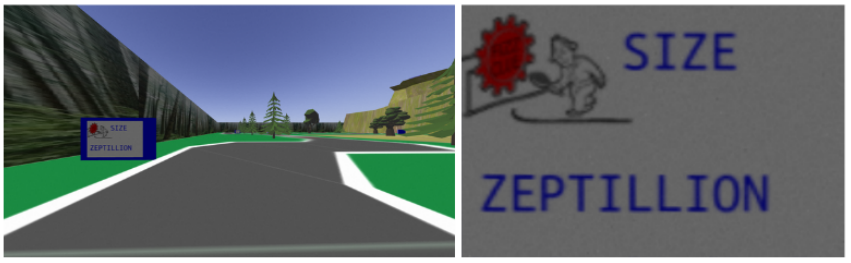


Fig 11 - The picture on the left is the camera image from the robot and the picture on the right is the output of "*get_banner_image*" in the ClueFinder class.

The function *get_banner_image* relies on many other functions that do some steps towards creating the perfect image. In this section of the report, we will briefly cover some of the main operations carried out on the camera image to get us to the perfect banner image.

To detect the location of the banner, we use a binary mask which captures the blue clue background. To ensure that we do not pick up banners too far away, a threshold on the area of the blue contour is used, and only process images that have more black pixels than the set threshold. Image below is the blue filter applied to the camera image:

Fig 12 - blue mask applied to the camera image

We then approximate a polygon around the largest contour of the binary image using *cv2.approxPolyDP* function. To make sure that we always approximate with a four-sided shape, we ensure that the length of the array returned by the aforementioned function is always equal to four. If not, we discard the approximation. Here is the approximated polygon drawn on the camera image:

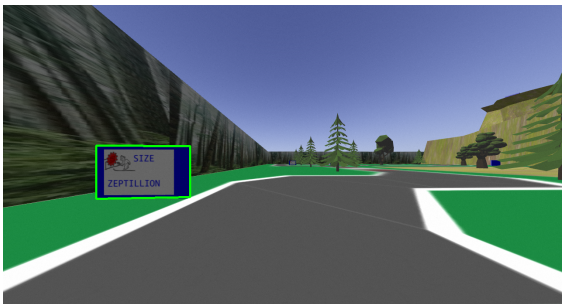


Fig 13 - green approximated polygon shown on the camera image

Fig 14 - The perspective transform of the banner

After finding the four corners of the banner using the approximated polygon, we perform a perspective transform on the region enclosed by the green lines in the figure above.

Image above is the output of the *find_banner* function. We also make sure the captured image has a high quality by setting a threshold on the blurriness of the image. We check the blurriness using the *is_blurry* function, implemented in the ImageProcessor class. Then, we remove the blue boundaries and only extract the white region. This operation includes creating a color mask, finding the approximated polygon, and performing yet another perspective transform:

Fig 15 - blue filter and bounding polygon on the banner image

Finally, this is a road map that illustrates how we acquire a well-cropped perfect image of the clue banner from the camera image of the robot. Note that the final image is then passed to the ClueGuesser class where the convolutional neural network guesses the clue topic and the clue value.

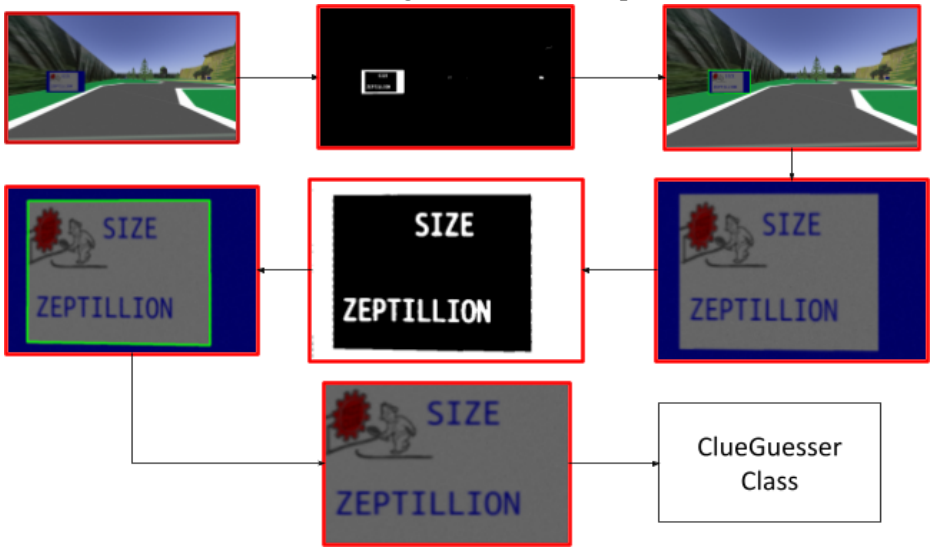


Fig 16 - Sequence of operations done in ClueFinder Class. Note that red boundaries are just added for better demonstration

Clue Guesser

Once we have a well-cropped clue banner image without blue boundaries, we pass it to the Clue Guesser class. Here, the CNN processes the image character by character, yielding a vector with 37 entries representing a probability distribution. We determine the predicted character by identifying the index with the maximum probability. To save time, we cross-reference the predicted topic with a list of possible topics (8 topics) before proceeding to predict the value, ensuring efficiency in the process

Score Keeper

The ScoreKeeper is a package with one main task: to communicate with the competition score tracer through the publishing to its node. The score keeper also keeps track of published clues and their topics which are used to determine which stage of the course the robot is at. The ScoreKeeper has the following methods: start(), end(), publish_clue()

PID Driver

The PID Driver Module is responsible for the navigation of the robot through the tracks of the course, keeping the robot within the road boundaries. Based on the segment of the course, the driver controls the robot separately.

1. Paved Road
2. Dirt Track
3. Desert (To travel from 6th clue to 7th in the off road section)
4. Mountain Road

We will first go through the steps which are common to all the segments.

The main computation to identify the road track was to generate a binary mask of the outer lines based on their HSV values of the pixels. The mask was created using the cv2.inRange() function. Below, you can see the masks created for the Paved vs Dirt vs Mountain segments

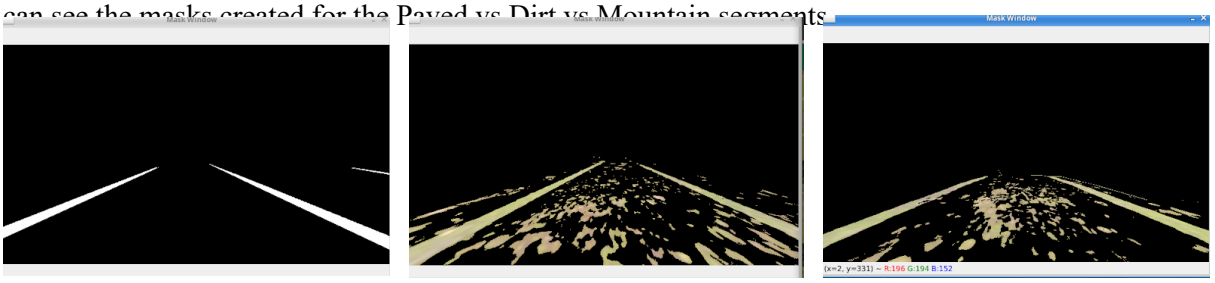


Figure 17 - Binary mask of road overlaid on color image. Paved - Dirt - Mountain segments

As observed above, the boundaries of the paved road, having its distinct white color, are identified immediately from the mask, as there are no other objects with that white color. We will consider the Paved road section first.

Now, to identify the road, we identify the two boundary lines of the road using the cv2.findContours() function. Then, we calculate the area of each contour and find its centroid using the cv2.findMoments(). A contour's moments are the weighted average of its pixels in the x and y directions and it represents the contours axial and rotational moments. This function allows us to find the center of mass of the contour. Once the center of mass of each contour is identified, we calculate the weighted midpoint.

$$C_x = \frac{Area(L) * L_x + Area(R) * R_x}{Area(L) + Area(R)} \;\&\; C_y = \frac{Area(L) * L_y + Area(R) * R_y}{Area(L) + Area(R)}$$

Fig 18 - Calculating the weighted midpoint of the road

We can see this applied to the image below. The red circles identify the center of mass of the boundaries, and the blue circle is the weighted midpoint.

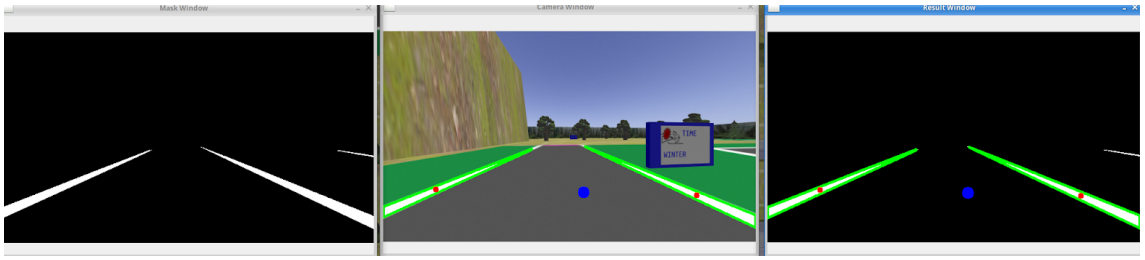


Figure 19 - The paved road identified using the contours and center of masses of the boundaries.

With the weighted midpoint, we can now calculate an error. Naturally, the error was calculated as the distance, in number of pixels, between the center of the frame and the center of the midpoint, visualized in blue, which gave the error in both the x and y directions.

$$e_{<x>} = MidPoint_{<x>} - FrameCenter_{<x>} \;\&\; e_{<y>} = MidPoint_{<y>} - FrameCenter_{<y>}$$

Fig 20 - The error calculations

To be able to drive the robot, we must give it an angular velocity and linear velocity. We experimented with a number of different approaches to calculate the angular velocity and linear velocities, and reached the optimization below:

$$v_{angular} = k_p * e_{<x>} + k_d * \frac{d}{dx} e_{<x>}$$

$$v_{linear} = v_{max} - (k_p * |e_{<x>}|)$$

Fig 21 - Calculating the velocity commands of the robot based on PID

Now going back to finding the boundaries in the dirt road sections, as observed in Figure ?, the masks for the dirt road are not as effective. To identify the correct contours as the We used three characteristics of the contours:

1. Area
   We first filtered the contours erasing all contours that had a smaller area than 250 pixels. But often, large patches in the middle of the road would surpass the area of the road side lines.
2. Solidity
   The solidity of a couture is defiance as the ratio of its area to its convex hull area. See Appendix E for more information. The line contours have higher solidity, so we chose a threshold for solidity of the contours.
3. Major Axis Length
   The road lines have larger major axis length, so we discarded contours with small major length
4. Location
   After filtering the contours based on We observed that most of the patches that are falsely identified are located in the middle of the road closest to the camera. So we defined a box where if a contours' center of mass fell within it, we removed it. The box is visualized below.

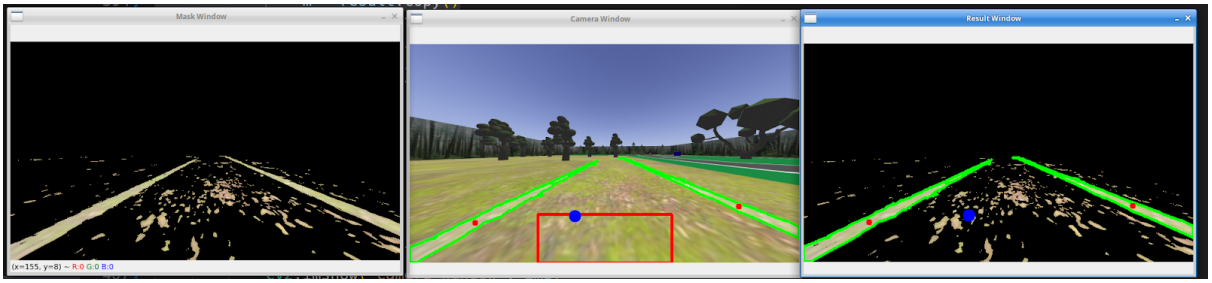


Fig 22 - The filtering of contours in the middle of the road based on their characteristics

Finally, at the top of the mountain, we located the clue board and PIDed on the clueboard until a valid clue detection guess was made.

Discussion

Safety Net

On rare occasions, our ClueGuesser would guess a clue correctly, but then overwrite that guess with an incorrect guess. To mitigate this problem, as a safety measure, we set a confidence threshold of 90% for all the predicted characters. If the probability of a predicted character falls below the confidence threshold, we disregard the entire image being predicted and wait for another image with better quality. Below is a graph of the confidence values of all characters predicted (both topic and value) during one whole round on the track:

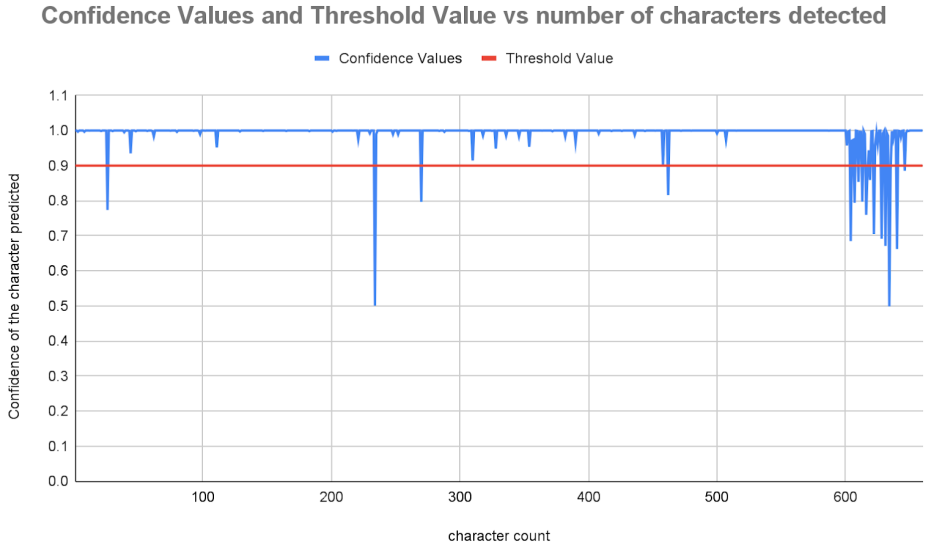


Fig 23 - The confidence of characters detected by the neural net

We can note that in some rare cases the confidence drops below 90% however, if that happens we do not publish the predicted character to the score tracker. Furthermore, since the confidence is mostly stable on 1 for the first 600 characters, we could also use a higher threshold confidence without leaving a clue unpredicted. Lastly, we notice that the last clue has a fairly lower confidence compared to the other clues. This is primarily due to the fact that the first few images captured of the last clue banner are too blurry and the CNN is not trained on them enough. However, as the robot gets closer and closer to the banner, the confidence goes up and eventually plateaus at 1.

Image Troubleshooting

To find the correct thresholding for different purposes, we used an interactive tool which allowed us to control the HSV thresholds while driving with the robot. This improved the accuracy of our thresholding and also saved us a lot of time.

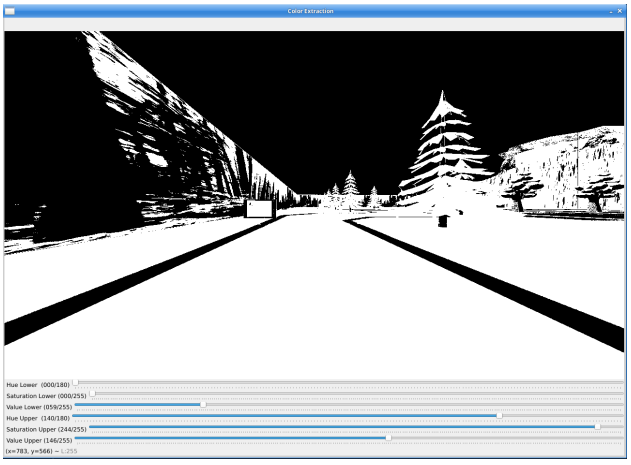


Fig 24 - Interactive tool used to find thresholding for different circumstances

Conclusion

The robot performed as expected during the competition and was able to secure 50/57 points. It detected 8 clues with 100% accuracy, and hence was awarded 52 points. 2 points were deducted since the robot was respawned from the second to the third pink line.

On the last day before the competition, we tried to implement code for navigating from the 2nd pink line to the 3rd pink line without the need for respawning. We were able to make it work, however, the challenge was to align ourselves perfectly with the tunnel so that we would not have to be worrying about PID inside the tunnel. We eventually did not adopt the implementation for navigation from the 2nd to the 3rd pink line. Nevertheless, after the competition we realized that we could just hit the tunnel and be awarded 5 points for reaching the tunnel and then respawn to the 3rd pink line. Had we known that this is a legitimate way of gaining the tunnel point, we would have definitely done it since we had the implementation already in place.

If we had more time, we would make the clue guessing process faster so that the robot could continuously execute the PID code without having to slow down to read the clues.

References

Thresholding Operations, OpenCV Documentations
https://docs.opencv.org/3.4/da/d97/tutorial_threshold_inRange.html

Contour Properties, OpenCV Documentations
https://docs.opencv.org/4.x/d1/d32/tutorial_py_contour_properties.html

Perspective Transform, YouTube
https://www.youtube.com/watch?v=Tm_7fGolVGE

Creating a Categorical Training Dataset from Directory, TensorFlow Docs
https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory

Convex Hull, OpenCV Documentations
https://docs.opencv.org/4.x/d7/d1d/tutorial_hull.html

Appendices

Appendix A: Convolutional Neural Network

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 98, 43, 32)        896

 max_pooling2d (MaxPooling2D  (None, 49, 21, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 47, 19, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 23, 9, 64)        0
 2D)

 conv2d_2 (Conv2D)           (None, 21, 7, 128)        73856

 max_pooling2d_2 (MaxPooling  (None, 10, 3, 128)       0
 2D)

 conv2d_3 (Conv2D)           (None, 8, 1, 128)         147584

 flatten (Flatten)           (None, 1024)              0

 dropout (Dropout)           (None, 1024)              0

 dense (Dense)               (None, 512)               524800

 dense_1 (Dense)             (None, 37)                18981

=================================================================
Total params: 784,613
Trainable params: 784,613
Non-trainable params: 0
_________________________________________________________________
```

Appendix B: is_blurry function

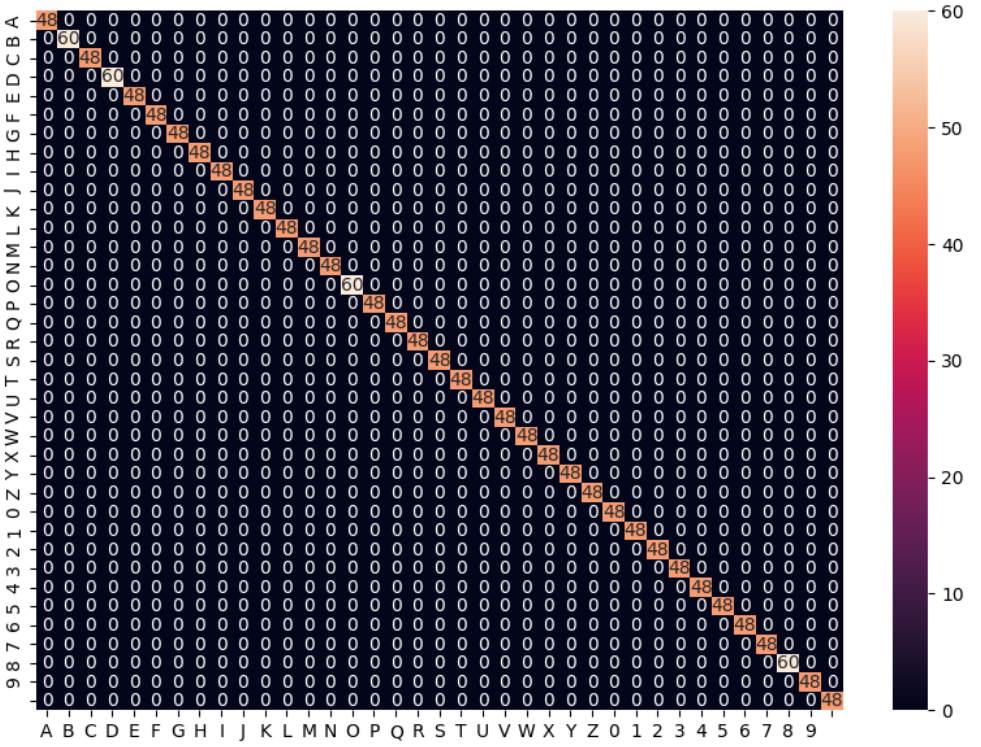
```
def is_blurry(image, fm_threshold):
    """!
        Determines if an image is blurry based on the variance of the  Laplacian filter.

    Parameters:
    image (numpy.ndarray): The input image.
    fm_threshold (float): The threshold value for the variance of the Laplacian filter.

    Returns:
    bool: True if the image is blurry, False otherwise.
    """

    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    fm = cv2.Laplacian(gray, cv2.CV_64F).var()

    if fm < fm_threshold:
        return True
    else:
        return False
```

Appendix C: Constants

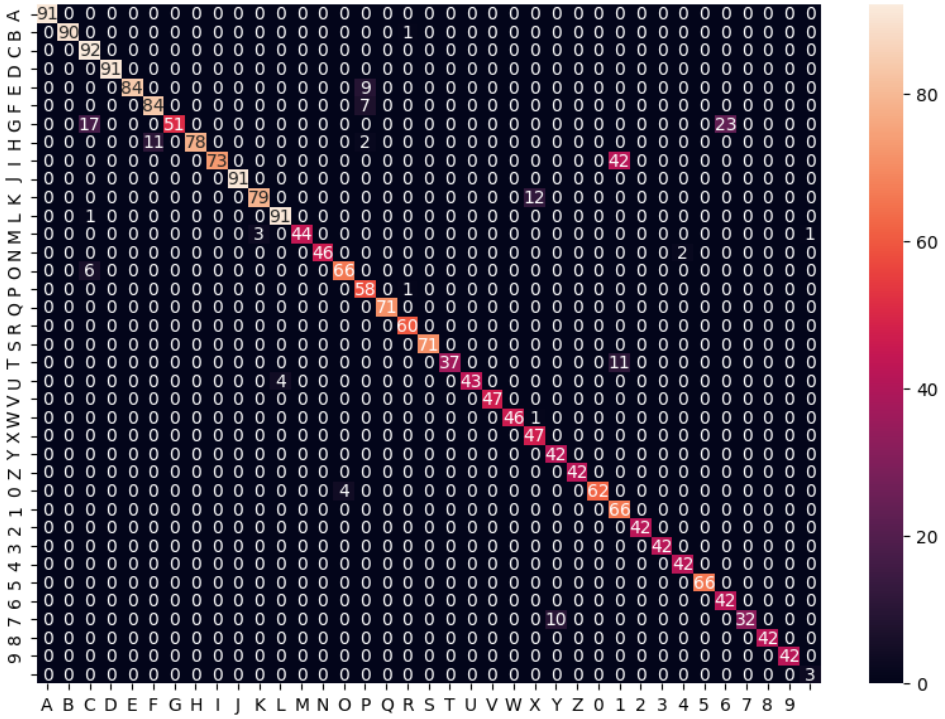
Constants for creating masks using HSV color filter						
Purpose	**Lower Bounds**			**Upper Bounds**		
	Hue	**Saturation**	**Value**	**Hue**	**Saturation**	**Value**
Blue Mask	120	50	50	150	255	255
Pink Mask	75	111	239	180	255	255
Red Mask	0	36	78	0	255	255
Pedestrian Mask	0	0	0	0	125	255
Truck Mask	0	0	0	180	255	5
Paved Road	0	0	220	5	10	255
Dirt Road	0	10	180	45	100	255
Mountain Road	120	160	170	180	230	240

Appendix D: Confusion Matrices

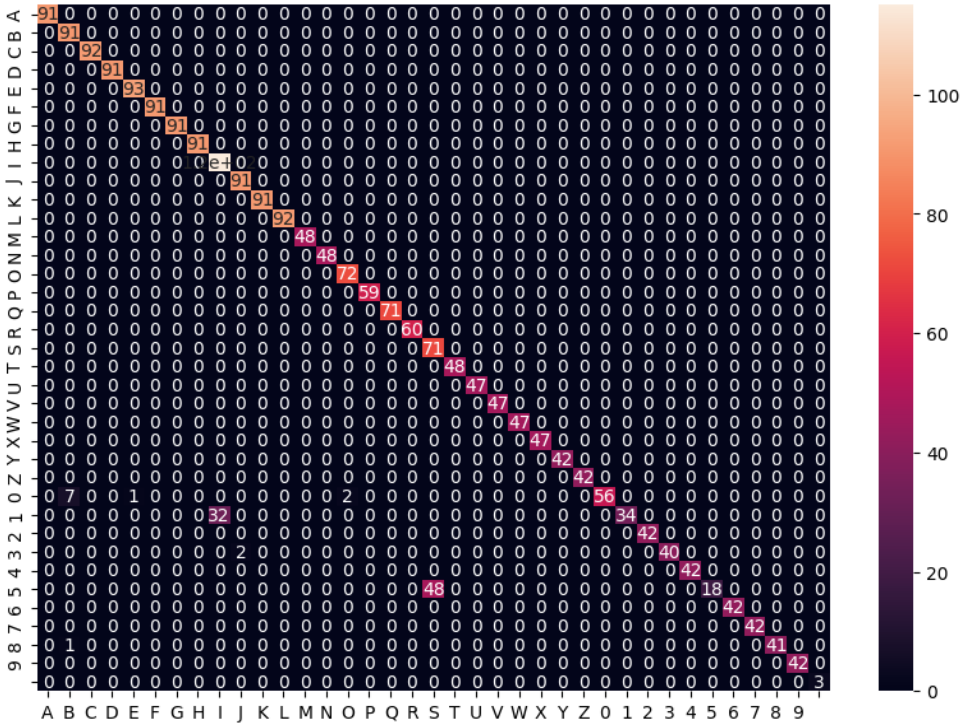
Below is the confusion matrix for the first round of training. In this confusion matrix, we see that all numbers are diagonal which indicates the high accuracy of the model. However, this is only for data that was artificially generated and the validation dataset does not contain any actual image from the competition environment.



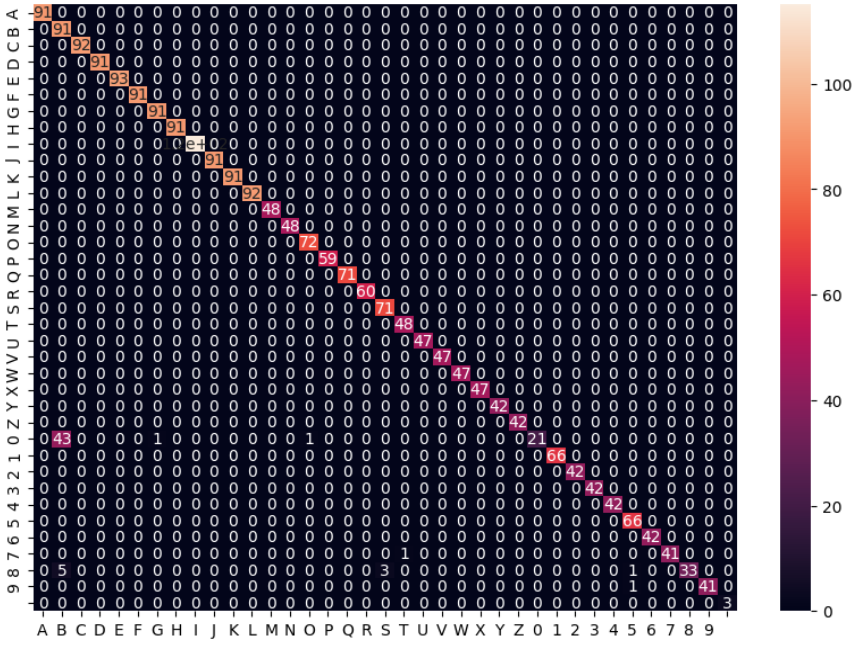
Below is the confusion matrix for the model that is only trained on artificial images generated by us (before the second round of training):



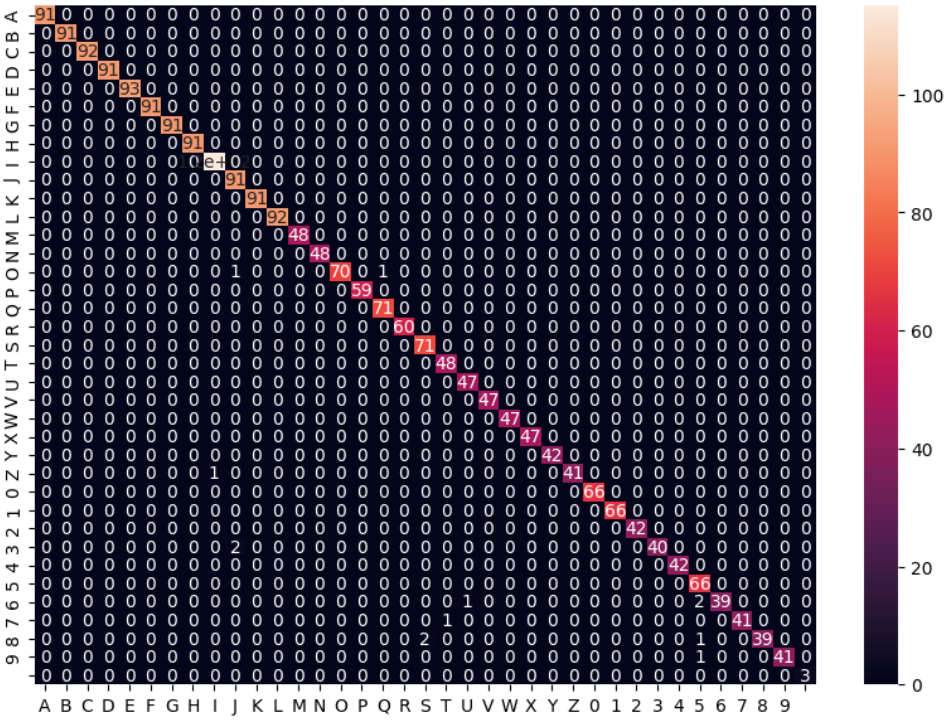
As we can see, the matrix is not perfectly diagonal and there are letter off diagonal such as 1 and I which the model has difficulty recognizing.
The confusion matrix below is for the model after the second round of training. We see that the confusion matrix improves, however, there is still some mix up between S and 5 as well as I and 1:



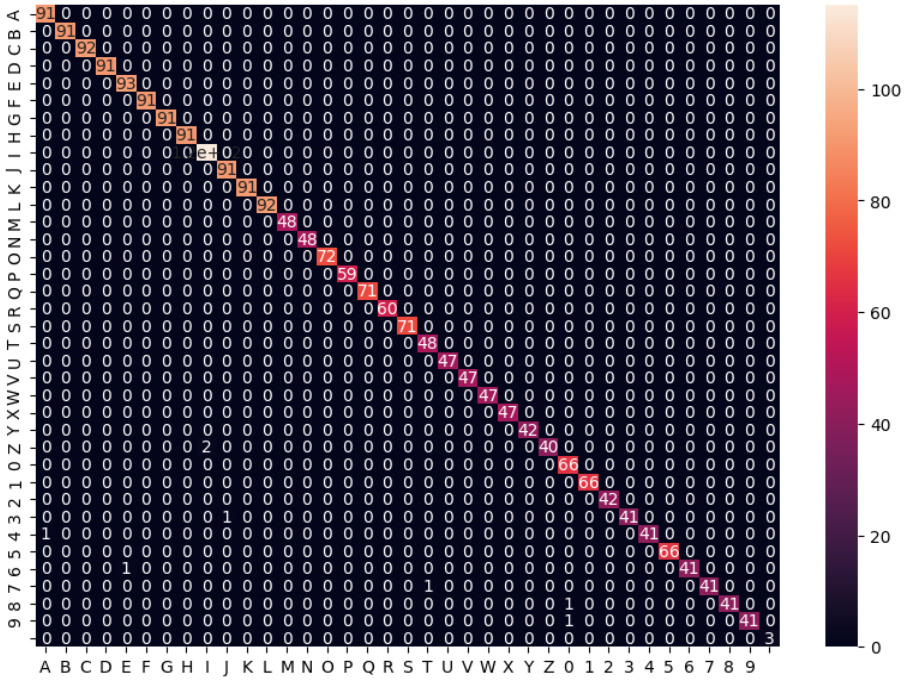
During the third round of training, the confusion matrix gradually started looking better and more diagonal:



After the third round was completed, the confusion matrix looked promising, with just a few sampled off diagonal:



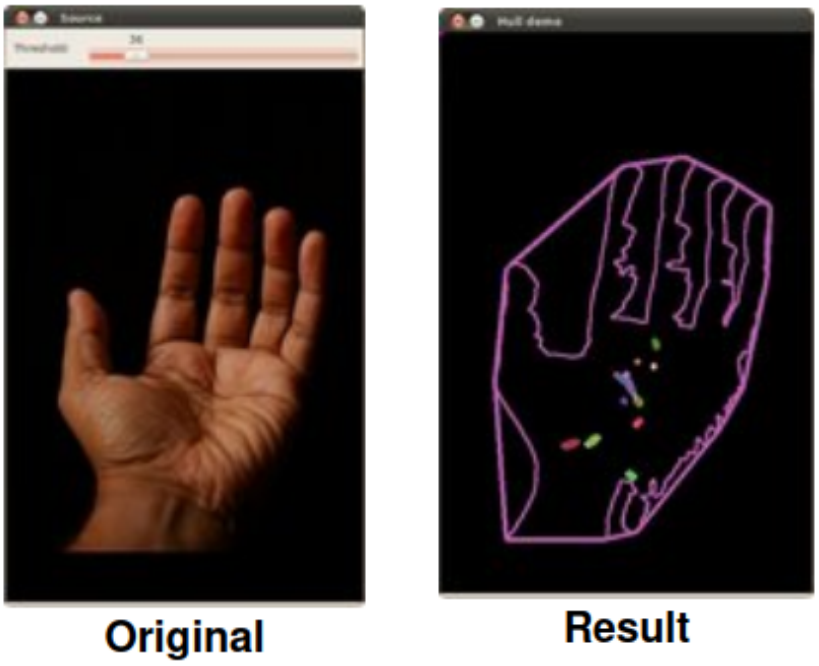
In the fourth round of training, we just brushed up on the Convolutional Neural Network model and just improved it on character specific datasets. Here is the confusion matrix of the final version of our model:



Appendix E: Contour Characteristics

Convex Hull Area:

The Convex Hull Area of a contour is defined as the area of the convex polygon that fully covers the contour. A convex polygon is one that has no concave angles.



Solidity:

3. Solidity

Solidity is the ratio of contour area to its convex hull area.

$$Solidity = \frac{Contour\ Area}{Convex\ Hull\ Area}$$

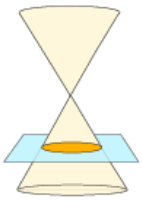
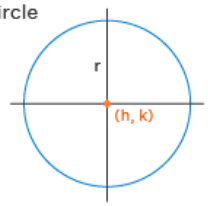
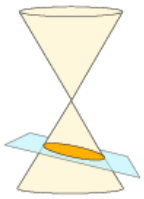
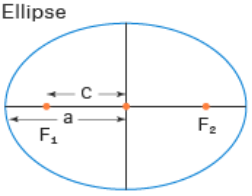
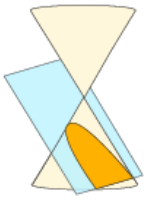
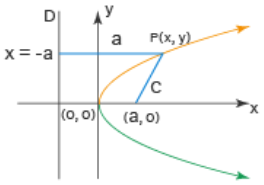
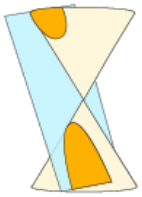
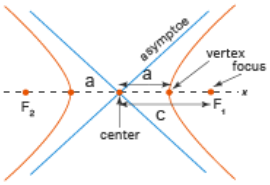
```
area = cv.contourArea(cnt)
hull = cv.convexHull(cnt)
hull_area = cv.contourArea(hull)
solidity = float(area)/hull_area
```

Eccentricity:

The eccentricity of a 2D shape is a measure of how circular the given shape is. In the context of our contours, we can estimate this by the ratio of the contours' major axis length and minor axis length. Thus, longer contours will have an eccentricity closer to

Eccentricity Formula

cuemath
THE MATH EXPERT

	Circle 	$(x - h)^2 + (y - k)^2 = r^2$	$e = 0$
	Ellipse 	$\frac{(x - h)^2}{a^2} + \frac{(y - k)^2}{b^2} = 1$	If $a > b$, $e = \frac{\sqrt{a^2 - b^2}}{a}$ If $b > a$, $e = \frac{\sqrt{b^2 - a^2}}{b}$
	Parabola 	$y = a(x - h)^2 + k$	$e = 1$
	Hyperbola 	$\frac{(x - h)^2}{a^2} + \frac{(y - k)^2}{b^2} = 1$	$e = \frac{\sqrt{a^2 + b^2}}{a}$