EEC 193 Autonomous Car Senior Design Report

Calvin Cramer, Daniel Loran, Victoria Salova
University of California, Davis

June 13, 2019

I. Summary

Lane line detection is one of the most fundamental systems of autonomous driving. Detecting lane lines serves as a basis module, on top of which path planning and other parts may be built. Knowing this, the lane line detection module must be robust, reliable, and accurate in order for these other systems to work. Even for non-self driving cars, lane line detection is the main component behind lane departure warning systems, and automatic steering in highway environments.

For this project, the first part was to combine two different versions of the lane line detection lab from Winter quarter and to find the speed and accuracy for use as a benchmark. The next goal was to find replacements for window search, such as using clustering algorithms, and also replacing one-dimensional window search with a two-dimensional search. The next part was to try and implement various deep learning models and see how they compare to classical computer vision approach.

II. Data Sets

i. Formatting CULane annotations - Calvin

The CULane dataset is a dataset of Beijing roads. The annotations included are for background, left-left lane, left lane, right lane, and right-right lane. Strangely, the annotations in the dataset are fixed width lanes from the car's perspective. An example of this can be seen in figure 1.

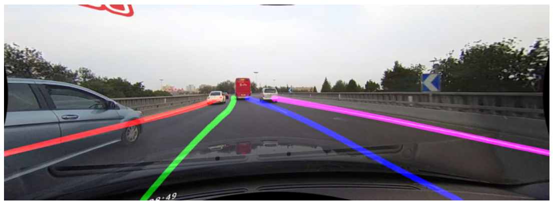


Figure 1: *Original CULane labels.*

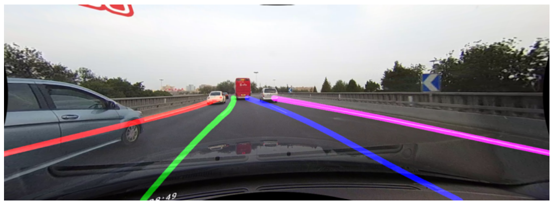


Figure 2: *Updated CULane labels.*

From the car's perspective lane lines start out thickest at the bottom and get thinner as the lane is further out. We decided to change these annotations to overlap the actual lane precisely, with tapering. The reformatted annotation for figure 1 is given in figure 2.

The purpose of reformatting the annotations like this is to train our deep learning models to detect just the lane, and not the surrounding space. An argument can be made to detect more of the lane in the distance, in order to give the model more of a chance to learn to detect the lane in the distance. Future work needs to be done to determine the validity of this, but post processing through dilation may help mitigate a small number of pixels for lanes in the distance.

Additional work can be done to round the end of the annotations, have a more reasonable drawing thickness for the left-left and right-right lanes, to determine the start and stop of each lane, and some tuning on the polynomial regression part of this reformatting procedure. Additionally, it may be useful to undistort the image into a top-down view, which may increase the accuracy of the output annotations.

ii. Scripts for CULane converstion - Calvin

The original annotations in CULane are given as one channel images with the values 0 for background, 1, 2, 3, and 4 for the left-left, left, right, and right-right lanes respectively. For the different deep learning models we used, all of them require the annotations to be in different formats, so we made scripts to convert the CULane annotations into these formats. An example of this is the the ICNet model requires RGB images, so the images were converted to RGB. Also, in order to actually see the annotations we mapped the 1, 2, 3, 4 values to high values, because if you open up the original annotations as a picture they look completely black.

Another script we wrote was to calculate the average RGB value for the whole CULane dataset train and test images. ICNet needed this average value in order to train properly.

Another script we wrote converted a directory of images into an avi video. The use of this script was to combine the output images from LaneNet into a video, since we didn't have a script that could automatically run LaneNet and convert the output images into a video, rather than just saving all of the images separately.

We also had a script to visualize the difference between the original CULane annotations and the reformatted annotations.

Another little script was to copy specific files (stored as paths in a file) to one location. The purpose of this was to gather specific test images from CULane that are representative of different driving conditions. CULane details many different driving scenarios, such as day, night, rain, no lanes, pedestrian, arrows, et. cetera. We gathered all of the images into one place (data/CULane/ folder in repo) in order to test out methods in these different scenarios.

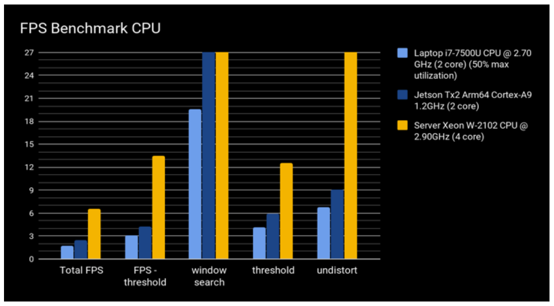
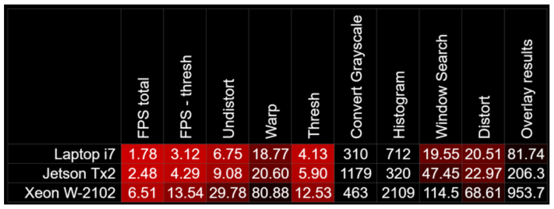


Figure 3: *FPS Benchmark CPU*



	FPS total	FPS - thresh	Undistort	Warp	Thresh	Convert Grayscale	Histogram	Window Search	Distort	Overlay results
Laptop i7	1.78	3.12	6.75	18.77	4.13	310	712	19.55	20.51	81.74
Jetson Tx2	2.48	4.29	9.08	20.60	5.90	1179	320	47.45	22.97	206.3
Xeon W-2102	6.51	13.54	29.78	80.88	12.53	463	2109	114.5	68.61	953.7

Figure 4: *Benchmark results*

III. Classical Computer Vision Approach

i. Benchmark for Lab 1 - Calvin

In order to explore different methods that work on top of the lab 1 pipeline, it was necessary to determine the most expensive parts of the pipeline. This allowed us to explore replacements for certain parts in the pipeline with the knowledge of how it would affect the running time of the pipeline. The benchmark measures the total times and frames per second for image distortion/warping operations, histogram, window search, waypoint generation, and a few others.

The benchmark was ran on three computers: Calvin's laptop, the Kronos server, and a Jetson TX2. The program ran completely on CPU, which is important to keep in mind when interpreting the results. The CPUs for these machines were an Intel i7-7500U (4 cores) capped at 50% max utilization, Jetson Tx2

Arm64 Cortex-A9 1.2GHz (2 cores), and an Intel Xeon W-2102 CPU @ 2.90GHz (4 cores) respectively.

Figure 3 and figure 4 show the results of the benchmark. The results of the benchmark show that common image manipulations are the bulk of the runtime. The main lane localization part (histogram, window search, polynomial regression) took a trivial amount of time. From this it is worthwhile to explore running the image manipulation parts of the pipeline on GPU, which is the type of operations GPUs are built for.

ii. Calculation of Radius of curvature and lane offset - Calvin

From the detected lane lines from the lab 1 pipeline (and any model where a polynomial of the lane is detected) we can calculate some important information that is vital for safety warning systems. The two pieces of information we calculated were radius of curvature 2 and lane offset 3d. The radius of curvature shows how straight or curved the detected lane is. A safety system may take this information into account, and can predict that if the driver turns to rapidly, that they may go past the lane. Lane offset explains itself nicely: it is the offset of the vehicle from the center of the lane. This information is a useful bit of information for lane departure warning systems. If we can detect that the lane is underneath the left or right wheels of the vehicle, and the driver is not changing lanes, then the driver may not be paying attention and drifting outside of the lane. Another use of this information is for a lane following system, where the system can try to minimize the offset from the center of the lane in order to automatically follow the center of the lane.

The calculation for the radius of curvature and the lane offset were fairly simple, all that was needed for the radius of curvature was to take the function for each lane and get the first 1b and second derivative 1c, and use a special formula. Note that the lanes are a function of y instead of x. The calculation for the offset of the vehicle in the lane is even simpler, using the assumption that the camera is in the very center of the vehicle. If this is the case, then the offset is just the pixel difference between the center of the detected lanes and the center of the video frame, multiplied by the radio between meters and pixels in the x direction. If the camera is not in the exact center of the vehicle, then this calculation can be calibrated by adding an offset to the "center-line" of the video frame.

$$f(x) = ax^2 + bx + c \tag{1a}$$

$$f'(x) = 2ax + b \tag{1b}$$

$$f''(x) = 2a \tag{1c}$$

$$RoC = \frac{[1 + f'(x)^2]^{\frac{3}{2}}}{f''(x)} \tag{2}$$

$$center = \frac{x_l + x_r}{2} \tag{3a}$$

$$x_l = lane_l(h_{window} - 1) \tag{3b}$$

$$x_r = lane_r(h_{windiw} - 1) \tag{3c}$$

$$offset = \frac{W_{window}}{2} - center \tag{3d}$$

iii. Replacements for window search - Daniel

The traditional computer vision approach to lane line detection is considered to simpler and less resource intensive than a neural network based approach. At the beginning of this project, a 5 stage modular pipeline (undistort image, RoI, thresholding, recursive window search, polyfit) was initially considered for modification. The recursive window search module was identified as an algorithm that could be replaced by a faster module due to its recursive nature.

Since clustering can be ran on all points in an image or bitmap, the motivation was to replace the recursive window search algorithm with a series of clustering algorithms, both to reduce noise and find lane lines in a non-recursive

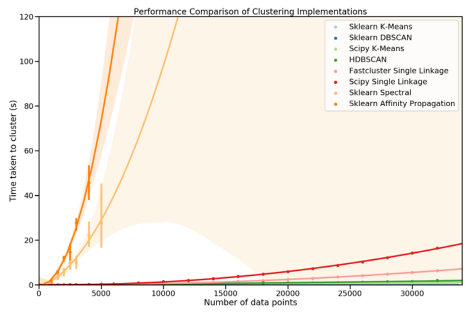


Figure 5: *Benchmark: data points vs time taken for different clustering algorithms*

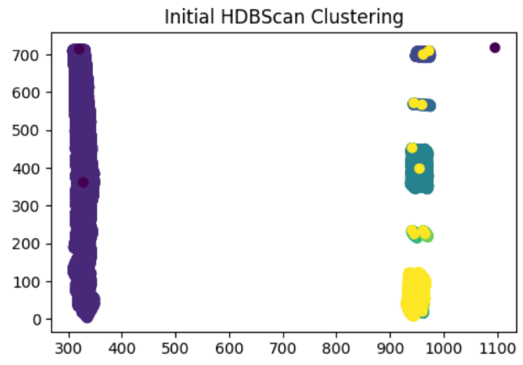


Figure 7: *Initial HDBscan clustering*

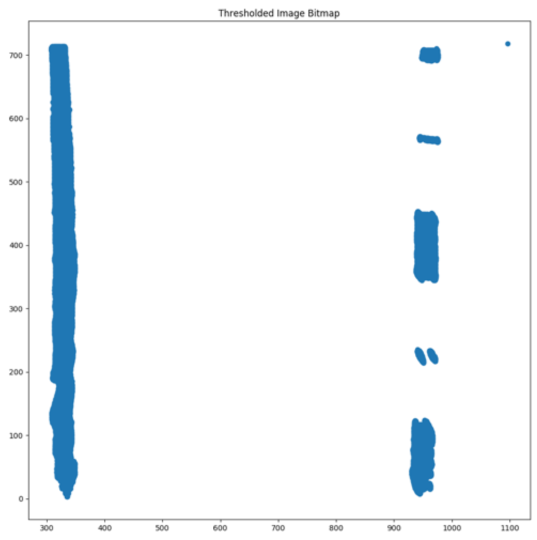


Figure 6: *Input bitmap to clustering module*

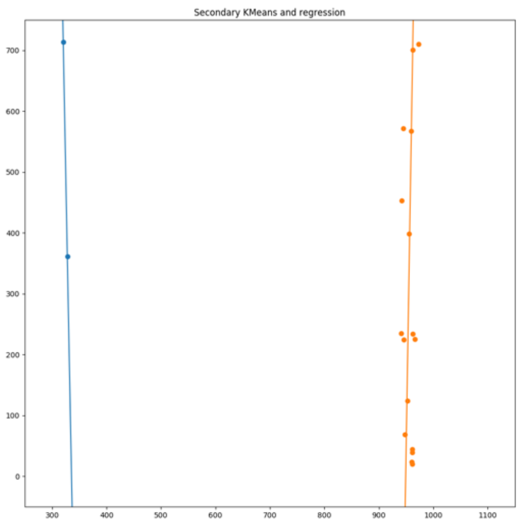


Figure 8: *Post linear regression*

fashion. In order to do this, multiple clustering algorithms needed to be tested. It can be seen in figure 5 the results of a benchmark test done on bitmaps of similar size and density to the ones that the previous thresholding step would produce. The sklearn k-means clustering algorithm stood out as a very fast clustering algorithm, although using kmeans would mean that the number of lanes must be known beforehand. HDBscan tests as a fairly slow algorithm but comes with the added functionality of identifying and filtering out points the algorithm deems noise.

The proposed clustering module has two steps. Initially, a bitmap is taken in from the thresholding module (figure 6). HDBscan is applied first to the bitmap to group points into initial clusters, while also removing noise (figure 7). Then, for each cluster a centroid point and a top y-value point are selected. After the centroid and y-value pairs are identified, they are clustered into left and right lanes using k-means clustering. After the k-means clustering, linear regression is run to estimate where the lines are (figure 8).

This clustering module ends up running at approximately 1.4 FPS, with limitations on what sort of noise it can handle in input images. Additionally, the clustering module is not able to handle curved lane lines, since it assumes a straight line is drawn between the centroid and high y-value pairing. HDBscan is most responsible for the low speed of this module. Changing parameters such as min_cluster_size and min_samples can result in a speed up but run the risk of decreasing accuracy. Future work on this module can be done in three areas - increasing resilience to noise, adding curve lane line prediction functionality, and increasing speed. Solving one of these problems might affect the solution of another problem; since HDBscan is used in part to remove noise, a new solution to noise in the bitmap might eliminate the need for HDBscan altogether, which might yield a speed increase.

iv. Initial foray into 2D Window search - Calvin

One additional avenue to improve upon the original pipeline is to make the window search algorithm more robust to highly curved lanes. In order to do this, we needed to overcome the implicit limitation of the one-dimensional window search, which is that it can only detect lanes at most at a 45 degree angle. For most driving circumstances this limitation is not an issue, but there are situations such at roundabouts or parking lots where it is necessary to detect sharp turns.

In order to turn the one-dimensional window search into two-dimensions, we need turn the displacement of the window from a one dimensional update in the y dimension into a displacement in any direction. Making the displacement into a vector brings a lot of uncertainty about when the algorithm will terminate. It is impossible to know the number of steps the algorithm will take beforehand, as opposed to one-dimensional window search which always terminates in a specified number of steps (which determines the window height).

We placed the following stopping conditions on the search:

- Terminate if the window reaches close to the left, top, or right of the image
- Terminate if the number of windows is greater than some specified amount

Another possible stopping conditions is to only allow the search to go "up" once, since we make the assumption that the lane lines start from the bottom of the image and that they curve nicely, generally in the upwards direction. It would not make much sense to see a lane line go up, and the back towards the car.

With these assumptions, we added a maximum angle change parameter, so that the search would smooth out, and not make drastic jumps in the angle of the displacement. Future work needs to be done to exploit this more, in the sense of using a PID controller to smooth the angle of displacement, and to give the search a general path or curve to follow when the current window doesn't have and lane line pixels, and must make a guess of where to go next based upon previous information. The guess we make on previous information now is simply to take the last displacement seen and continue in that direction. The PID controller may also help with oscillations that arise in this method. Since we are taking a step in the direction of where we see the lane line pixels, if we start a little bit off-center from the lane line at the bottom of the image then the algorithm may follow the lane but oscillate from the left of the lane to the right of the lane. In order to mitigate this we centered the first window with the nearby pixels, rather than starting from the histogram peak which may not be a good starting point for the lane.

Additional future work needs to be done to ensure that lane line points that are detects as "in the lane" are not double counted. This problem arises in 2D window search because the displacement radius is a tunable parameter, which may lead to successive windows overlapping. This problem is non-existent in one-dimensional window search because the displacement of the windows makes it impossible for the windows to overlap. Additional future work includes tuning the input parameters for this method. The parameters include maximum angle change, displacement radius, window width, window height, maximum number of windows, and the minimum number of pix-

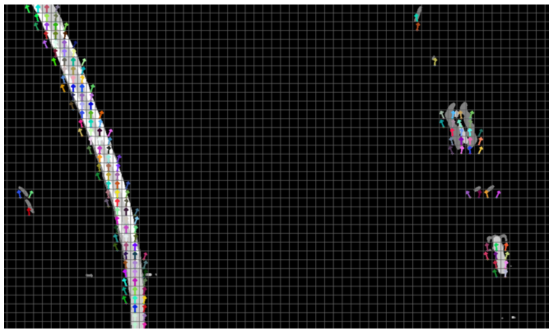


Figure 9: *Vectors*

els needed to consider a window signal (not noise).

v. Comparison between 1D and 2D window search - Calvin

The goal of the two-dimensional window search is to perform better on tight curves than the one-dimensional window search. This needs to be tested to see the difference between the two methods. In particular, the difference should be tested in various video scenarios.

vi. Experimentation with vector field (see hough branch) - Calvin

We originally started another lane line detection pipeline by converting the thresholded bitmap image to a vector field. The process of this was to split the image into a grid, and for each cell make a vector between the bottom-center of the cell and the average point of the top-most n rows of the cell. Although this part was not used in any pipeline, it's idea is fairly similar to the two-dimensional window search, and further work needs to be done to incorporate this in a complete pipeline or to incorporate the vector field in a two-dimensional search. An example of the vector field is included in figure 9.

IV. Deep Learning Approaches

i. SCNN - Victoria

The first neural net we looked at was Spatial CNN. It is a generalization of deep convolutional to a spatial level and it outperforms recurrent neural nets, Markov random fields, and conditional Random field, by both speed and accuracy. SCNN views rows and columns of feature maps as layers and applies convolution and nonlinear activation sequentially, forming a deep neural network. The advantage of SCNN over other lane line detection methods is the fact that sequential message pass scheme is more computationally efficient than other RNN or Markov Random Field approaches - it gets rid of computational redundancy that other models have. It utilizes sequential propagation scheme instead of each node receiving messages from all others. So information goes from right to left and top to bottom, so 4 directions, which results in the same amount of information to be passed, but with less computation. SCNN can also be incorporated into any part of CNN, rather than just the output. The ablation study that was done on SCNN shows that every part of this network is important and not redundant. SCNN distinguishes each lane line as its own class, creates a probability map for each, which give prediction on the existence of the lane line markings. After that, these predictions are connected by cubic splines to create the final predictions of the lane lines. SCNN was compared to ReNet, MRFNet, DenseCRF, and the very deep residual network. Out of all the methods listed, SCNN performed significantly better, showing the best results.

In order to get this model running, we spent a significant amount of time setting up a proper docker container that would allow us to use the GPU for training. We also edited parts of the training script, updating the outdated tensorflow functions as well as editing the scripts to train on CULane dataset. Unfortunately, we decided not to proceed with this model as we realized we did not have the hardware to train this model. The last error that we ran into was GPU running out of memory. The attempts to decrease the batch size and other configuration settings to decrease the memory used turned out unsuccessful and we decided not to proceed with this model.

ii. LaneNet post-processing - Calvin

The purpose of post-processing the output of LaneNet to fit polynomial lines to the lane predictions of LaneNet. Additionally, the output of LaneNet would swap the annotation class of the lane lines frequently, so an additional task here is to give each lane a consistent class, according to their location on the image.

Post-processing is done by taking each unique class in the input image, and doing polynomial regression to draw a smooth lane. The output of the post-processing were not good enough at this point, and can be improved by undistorting the image, and doing polynomial regression with for a lower order polynomial, and possibly dilating the input image to have more pixels to do regression on. Additionally, the RoC and lane-offset parts can be included here.

The way that we gave lane lines a consistent class was so sort the centers of the lane lines according to the angle from the center top of the frame. This gave fairly good results, but the classes were not 100 percent consistent in our output video. The reason for this is that sometimes a lane may not be present, and the algorithm assumes that all of the lanes will be present. A simple check to see if each lane line class is present or not may solve this problem.

iii. ICNet - Victoria

The next deep learning approach we looked at was ICNet. The reason for this choice was that ICNet is one of the only semantic segmentation models that can run close to real time. The way ICNet works is that it downsamples a large input image by 1/2 and then by 1/4. The smallest image is ran through a heavy CNN and the two smaller ones are ran through a light CNN. A cascade feature fusion unit is then used to combine the results of the images

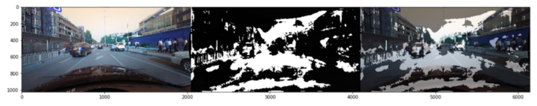


Figure 10: *First ICNet training resuts.*

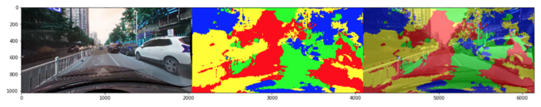


Figure 11: *Smallest image weighted the most.*

to refine the segmentation in the higher resolution image. The reason ICNet is able to run so quickly while maintaining accuracy is because a heavy CNN is ran only on the smallest image, taking significantly less time than if it were run on the original image, so the run time is shortened significantly.

Just as with SCNN, the set up took a significant amount of time. The biggest challenge was finding the correct docker image that would allow us to train our model on the GPU. The model for the code that we used and its documentation were outdated, so we had to adapt the code to run on our systems. Most of the initial updates were debugging and replacing the outdated tensorflow functions.

Since ICNet was designed as a semantic segmentation model for scene understanding and trained on Cityscapes dataset, we couldn't use the pretrained weights and had to train the model from scratch. On average, it took about 14 hours to run the 60000 steps to train the model. The first run did not produce any results and just classified the entire image as background. Since we are trying to convert this model to predict lane line (which is a semantic segmentation subproblem) and are using the CULane dataset instead of Cityscapes, we had to refactor some parts of the training

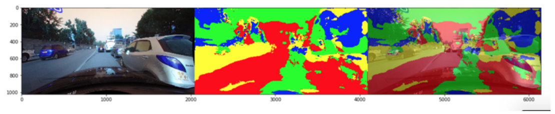


Figure 12: *Image downsized by 1/2 is weighted the most.*

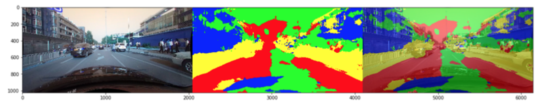


Figure 13: *The original image is weighted the most.*

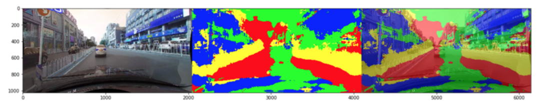


Figure 14: *The image with the best weights.*

and evaluating scripts. First step was to relabel the CULane labeled images to have RGB labels, instead of grayscale. Another step was also to change the labels in the model. The CULane dataset consists of 5 labels – 4 lane lines and background. Each lane line from left to right has its own class and there could be up to 4 lane lines labeled in each image (figure 16). The labels also go through the obstacles in the image, such as cars, and are drawn over that, even if the actual lane line is not present in the image. After that, we trained the model for the first time using the arbitrary parameters that were used in the original ICNet implementation. After the first training, the evaluation showed 19% IoU and the result in figure 10 shows only one of the labels, but it is very clear that the label is all over the place.

For the next step, we tried adjusting weights differently for each of the images, getting slightly better results, but still needing more tuning, figures 11, 12, 13.

The figures 14 and 15 use the same weights, but figure 15 has a less occluded environment. The IoU for this iteration was about 86%. In the figures, the label colors do not match exactly. But it is obvious in the last image, that the labels are encompassing the entire lane lines, and the surrounding areas as well, which is

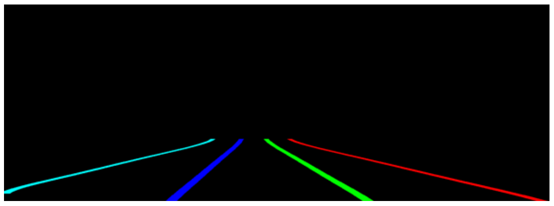


Figure 16: *Original Labels*

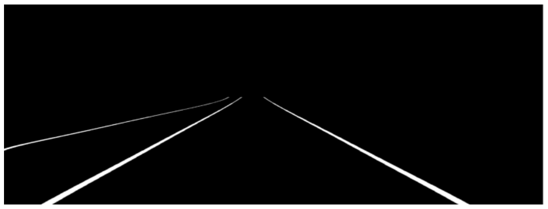


Figure 17: *Binary Labels*

a problem throughout all the iterations above — the entire image is labeled using the labels, ignoring the background completely. As we looked through the code, there was an ignore label in configurations. Eliminating that variable, however, did not change the results, neither did adding a dummy label and setting the ignore label to this dummy label. To fix this problem, we once again relabeled the dataset, figure 17, but this time we decided to do just a binary classification: lane line and not a lane line. This was the last test we could run before the deadline, and the results were unsatisfactory, where the entire image was labeled as one class, figure 18.

The future steps to solve this problem would be to try to figure out exactly why is the model ignoring one of the labels and change how the model weights background pixels and label pixels, and make sure that the label pixels are weighted slightly more than the background.

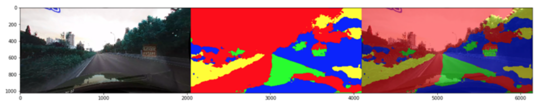


Figure 15: *The image with less occlusions.*

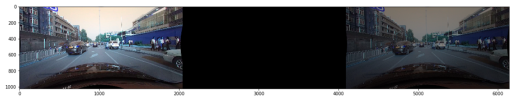


Figure 18: *Image trained with binary labels.*

V. Conclusion

At the beginning of the quarter, we set a few goals regarding this project. The very first steps were to combine the code from Lab 1 (Victoria) and then find out the speed and accuracy of that code to use as benchmark (Calvin). This gave us a basic structure to improve upon. We then explored different clustering alogrithms and replaced the window search with DBScan clustering technique (Daniel). Our next goals included benchmarking deep learning models, such as SCNN, LaneNet, and ICNet, however, we ran into difficulties running those models. We tried running SCNN and spent most of the time fixing ICNet to work as a lane line detection model (Victoria), and we also looked at the LaneNet model (Calvin). We were also planning on recording our own video and testing the lane line detection models we came up with on that video, but unfortunately we ran into some set backs along the way and did not accomplish this goal.