Lane Detection Technical Report

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

Shao-Yu Weng
nthu106071077@gapp.nthu.edu.tw

Bo-Yu Cheng
nemo1999.eecs06@g2.nctu.edu.tw

June 28, 2021

The Demo Video can be found at https://youtu.be/9Lgm3DqLSRM

1 Introduction

Lane detection, as the name suggests, is detecting lanes on the road. Currently, lane detection has been used in lane departure warning systems (LDWS). After learning many topics and techniques in computer vision this semester, we want to make something related to some of the topics and compare the results of deep learning methods and the traditional method.

For the traditional method, we find two different pipelines that can detect lane lines with acceptable accuracy. The main contribution of our traditional method is that we successfully detect more than 2 lane lines, which haven't been tried using traditional CV methods before, to the best of our knowledge. Our algorithms try to detect 4 lane lines in all images, and achieve about 70% accuracy on Tusimple dataset.

For the deep learning method, we apply two hourglass network with lane loss and double hinge loss for lane detection. This method showed 97% accuracy on the TuSimple dataset.

Overall, the contributions of our work are as follows:

- We modify two traditional methods and successfully detect more than 2 lanes with accuracy over 70%.
- We reach high accuracy by apply hourglass network and double hinge loss.
- Compare the accuracy and show the results between three methods.

2 Dataset

Before going on to the actual implementation, we will introduce the dataset we are using for training and evaluation. We use TuSimple as our dataset. The TuSimple dataset consists of 6408 road images on US highways. The resolution of images is 1280x720. The dataset comprises 3626 for training, 358 for validation, and 2782 for testing called the TuSimple test set, of which the photos are under different weather conditions.

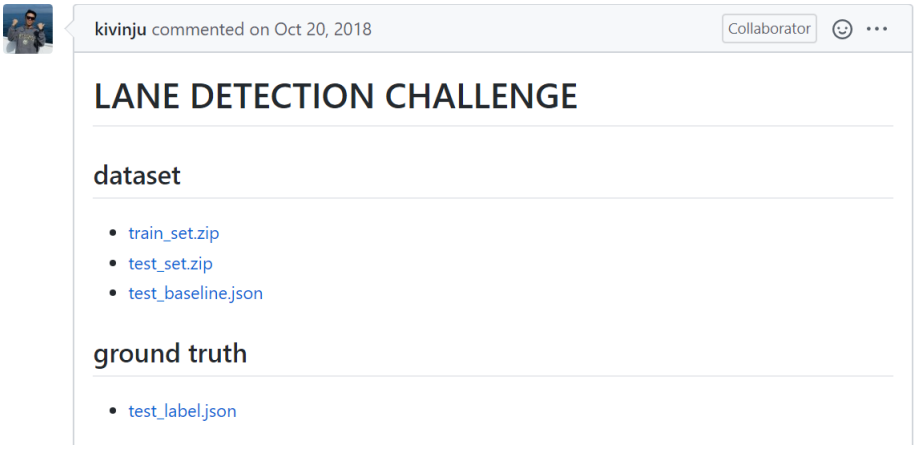


Figure 1: Tusimple-benchmark website page.

Num	Training set	Testing set
Five	239	569
Four	2982	468
Three	44	1740
Two	1	5
Total	3626	2782

Table 1: Lane number corresponding to the number of data in training and testing set

We download the dataset from Tusimple-benchmark. There is a JSON file in every clip folder containing the ground truth lane coordinates in that specific short clip (20 consecutive images). We split the data by its number.

2.1 Data preprocessing

For traditional methods, we only use dataset for testing. However, deep learning method will need to preprocess the dataset. We have made some changes to each image as data augmentation in the preprocessing step. For each image, we have done the changes as follow:

- Random flip.
- Random translation (affine transformation).
- Random rotation.
- Add random Gaussian noise.
- Random change the intensity of the image.
- Add random shadow to the image.

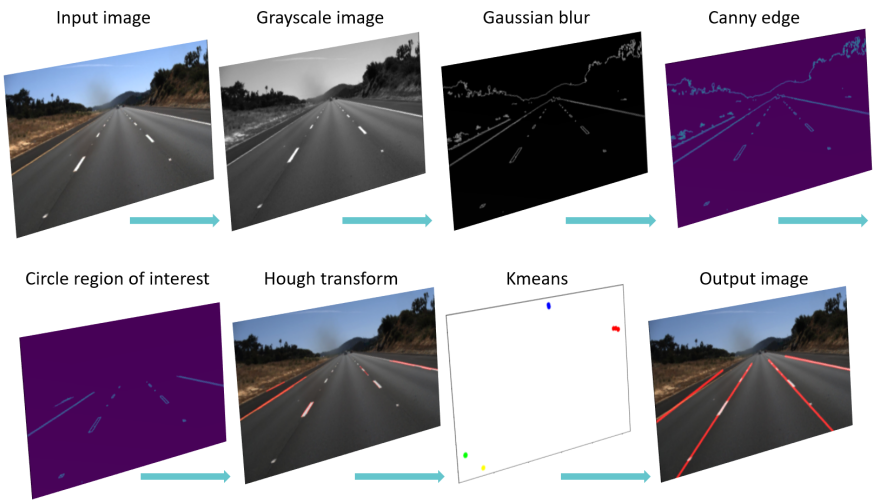
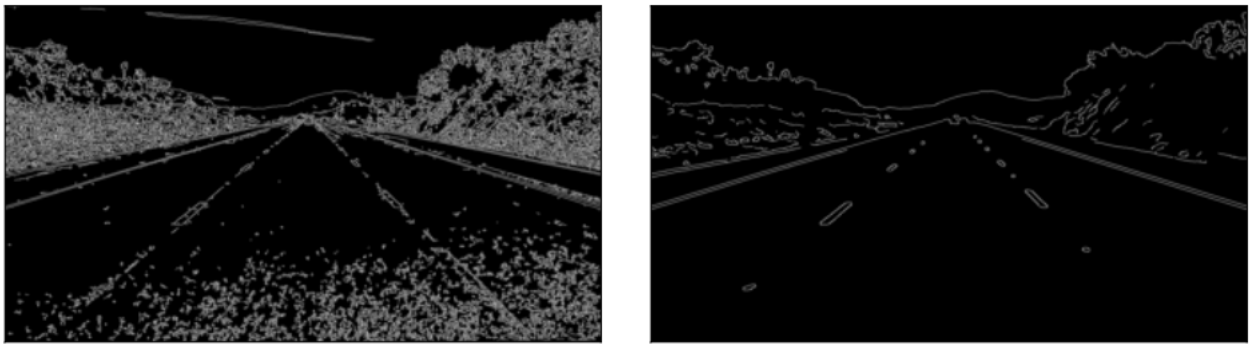


Figure 2: Pipeline of Method 1



(a) Without Smoothing

(b) With Smoothing (kernel size=15)

Figure 3: Effect of Gaussian blur

3 Traditional methods

3.1 Method 1 - Edge detection + Hough transform

The main idea of this method is utilizing Hough Line transform and K-means algorithm to extract and seperate different lane lines in the image. The idea of using Hough Line comes from Tutorial: Build a lane detector, while using K-means is our origianl attempt. We show the whole pipeline in Figure 2.

3.1.1 Canny edge

The first insight we get is that edges of lane lines tends to have stronger gradient than other edges, so we use a low pass filter to reduce unwanted noise edges. We use large gaussian filter (kernel size=15) to smooth the image before Canny Edge detector. We do another gaussian blur (kernel size=3) after Canny edge, to make the edge lines thicker, which help succeeding steps.(Similar results be achieved by adjusting thresholds of Canny detector)

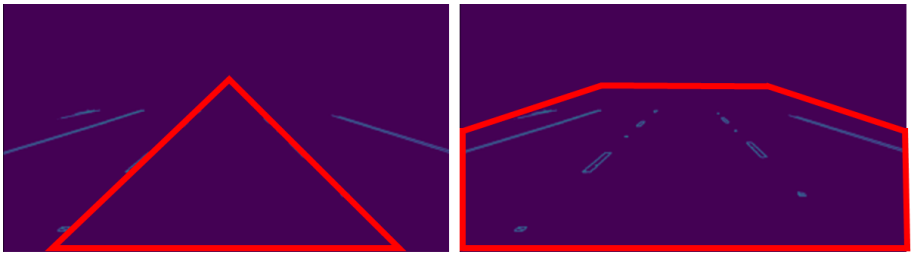


Figure 4: Comparison our region of interest with others

Canny edge find edge by looking for local maxima of the gradient $f(x, y)$. This method set two thresholds to select strong edge and weak edge. Strong edge is what definitely be an edge while weak edge is candidate for the edge. Here we set high threshold to be 20, low threshold to be 50.

3.1.2 Region of interest

Since we only concern about the road, we fix our perspective into a certain region to get rid of unrelated noise. Most of previous work just try to find out the closet two lanes to car, therefore they set their region of interest into a triangular. However, in order to compare with deep learning method, which can draw more than two lanes in the same time, we define our region of interest into a polygon, shown in Figure 4. In this way, we are able to locate more lanes yet it also brings us more interference sometimes.

3.1.3 Hough transform

Hough transform is a feature extraction technique. It can identify the geometric shape in the image through projecting points into Hough space then voting.

The reason to use Hough transform is that we assume lane lines edges to be straight on the image. While the shape of cars and mountains are round and curved.

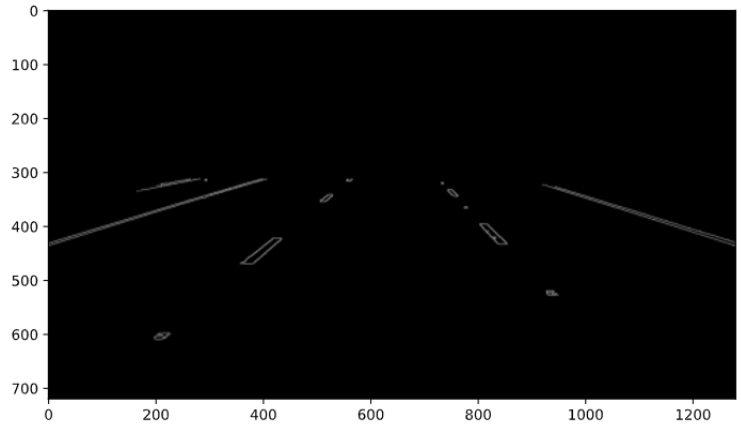
We use the build-in function in OpenCV "cv2.HoughLinesP", and adjust parameter ρ and θ for lower resolution on Hough space, which makes the line detector more sensitive. We also make "max_line_gap" higher, which allows us to merge "Dotted White Lines" together and detect the segments as a single straight line. Demonstrate in Figure 5.

3.1.4 Kmeans

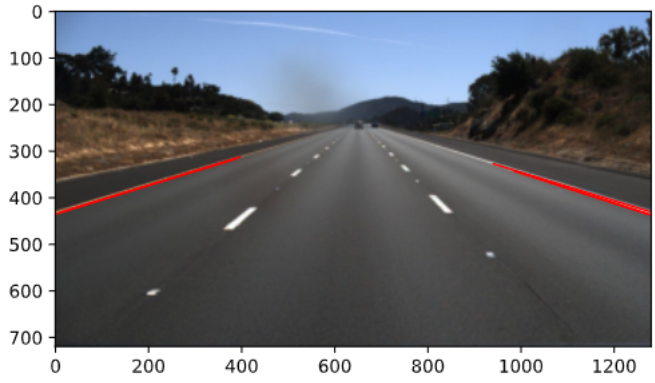
As mentioned above, since most of the existing works only focus on detecting two lanes, they classify the line segments into left and right according to the sign of slopes. There are two problems about the previous methods:

1. First, we can not locate more than two lanes.
2. Sometimes our perspective will deviate, causing the slopes of left and right lanes having the same sign.

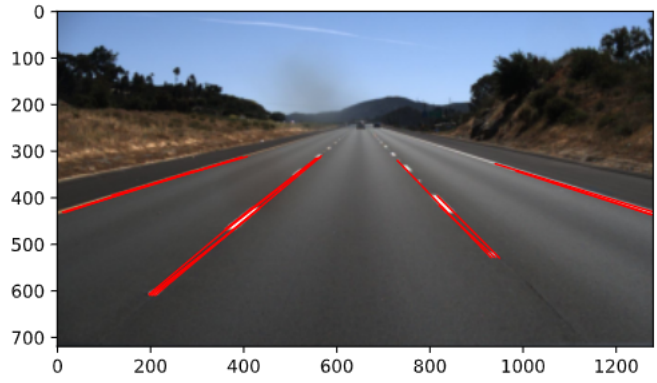
We fix these problems by clustering the line segments with K-means algorithm. K-means is a cluster algorithm, provided the number of groups k, it will randomly choose k cluster centroid, and assign



(a) Input for Hough transform



(b) max_line_gap = 50



(c) max_line_gap = 300

Figure 5: Effect of max_line_gap

each observation to the cluster with the nearest centroid. It then recalculate the centroid and class assignment alternately until convergence.

In addition to slope, we add the x-intercept of each line segment as second feature of K-means. Since most of the pictures in Tusimple dataset has 4 or more lanes, we set k=4 here. Result is shown as Figure 6.

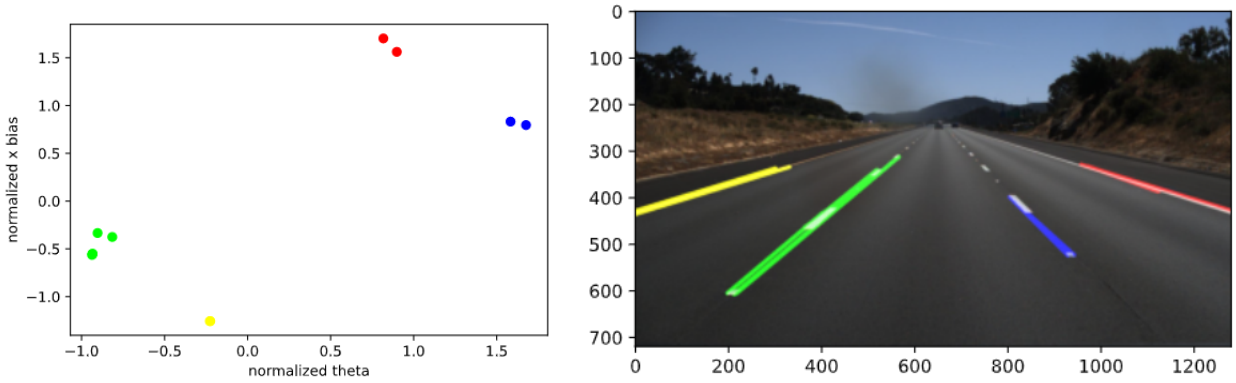
3.1.5 Curve Fitting

Finally, for each class of line segments, we collect their endpoints, then fit the points with $x = f(y)$, in least square error. Here we set $f(y)$ to be a polynomial of degree 1. Finally, we can draw the resulting lane line in a fixed y-interval(same as the ROI region), result is shown in Figure 7 .

3.2 Method 2 - Perspective transform

Key concept of this method is perspective transform. We warp the input image such that the lane lines are verticle in the warped image. In other words, we change the perspective of the image to bird-eye view.

There are 2 advantages of this warped perspective coordinate:



(a) K-means feature space

(b) Line segments Cluster result

Figure 6: Result of Kmeans

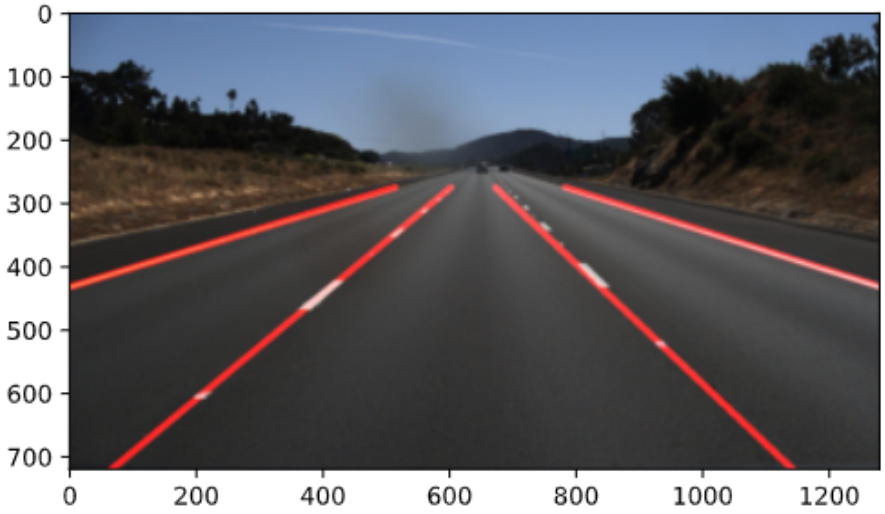


Figure 7: Result for Method 1

1. Seperation of different lane lines is very easy: points of different lanes has different x values.
2. Extracting edges that belongs to a particular lane line is easier, because they tends to have the same x values.

After we finds out the lane lines, we transform it back to original coordinate system.

The idea of using perspective transform comes from Automatic Addison, we change several stages of their pipeline to achive more stable result. The pipeline of Method 2 is shown in Figure 8.

3.2.1 Perspective transform

We warp the camera perspective into bird-eye view perspective by cv2.getPerspectiveTransform. Define our region of interest(ROI) and our desired region of interest(DROI) we are able to get transformation matrix and inverse transformation matrix. Transformation matrix help us warp image into new coordinate, while inverse transformation matrix let us to combine the result of lanes

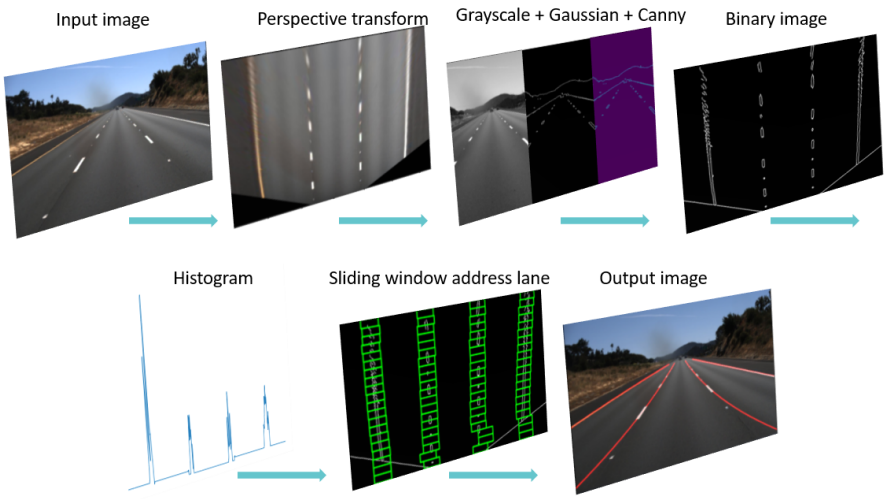


Figure 8: Pipeline of Method 2

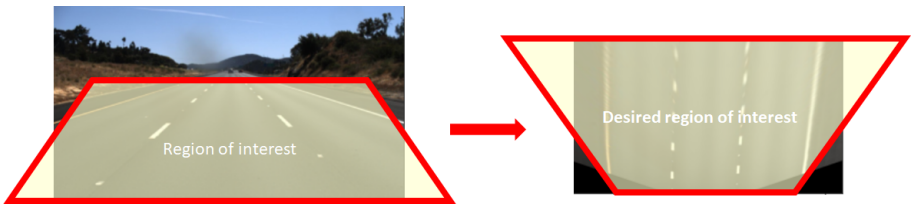


Figure 9: Region of interest v.s Desired region of interest

with the original image. There are one things to notice, since ROI and DROI decide the quality of transformation matrix, we finetune the hyperparameters until it can get reasonable lanes. For example, we set our ROI and DROI as Figure 9.

3.2.2 Garyscale, Gaussian blur and Canny edge

In Automatic Addison, they detect edges using color masking and intensity thresholding , also, the edge detection is performed **before** perspective transform . After many experiments, we find out that doing edge detection with Canny Edge detector **after** perspective transform gives much more stable result. See second and third points of Discussion for more detail.

We do gaussian blur before and after Canny Edge detection as in Method 1. The parameter settings are also the same.

3.2.3 Histogram

First, we apply cv2.threshold to make our image binary. After that, we sum the number of nonzero pixels along vertical direction for each x values, which help us to build up the histogram. As shown in Figure 10, the peak of the histogram means the density of edge is high, that is, there might exist a lane.

So far, we are assuming that the lanes are all straight, and that the camera is facing a direction that is parallel to the lanes, now if the the lane has large curvature, or the car is moving across the lanes, then the lines won't be perfectly vertical in the warped coordinate. In these case, the histogram

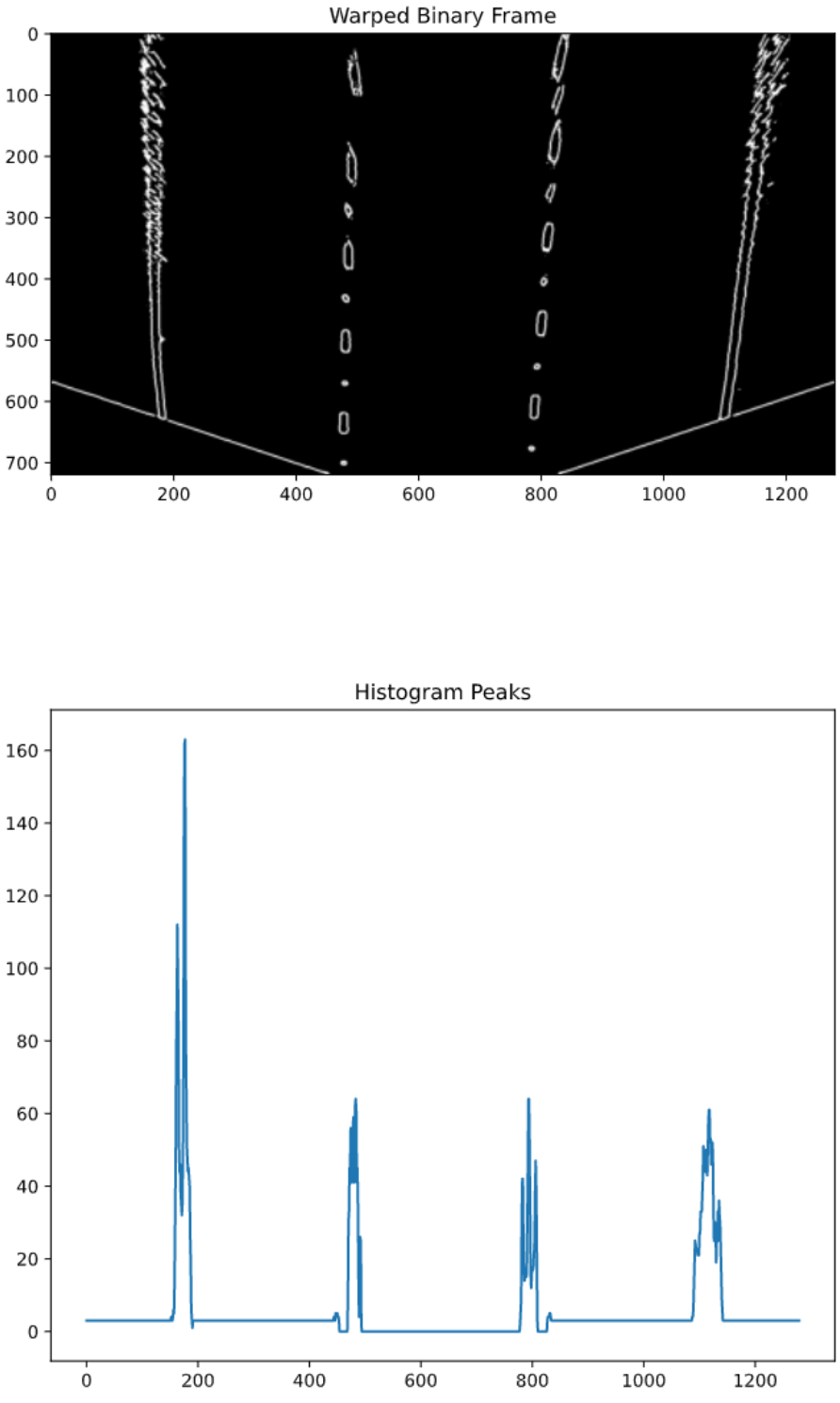


Figure 10: Histogram peak and corresponding binary image

may be blurred or less pointed (as shown in Figure 11 (a)). We figure out that calculating the histogram using only the bottom portion of the image gives more stable result and partially solve this issue(see Figure 11 (b)).

3.2.4 Sliding window

The peaks of histogram are used as the initial x-values of sliding window algorithm. The steps of the algorithm shown in Algorithm 1.

When the line tilt or turn, the window can adjust its position and follows along the lane. Note that it would be very hard to trace the lane line in the original coordinate, because (1) Lane lines would squeeze together at the horizon (2) Lane line would be so tilted that the algorithm will fail.

Algorithm 1 Sliding window

1: Set the initial x as the peaks of the histogram.
2: Initial y is set to the bottom of the image
3: Calculate the number of non-zero pixels in the window area $[x-m, x+m] \times [y, y+\frac{h}{\#windows}]$ of binary edge image, where m is the margin , h is the height of the image, $\#windows$ is the number of sliding window that we are going to use.
4: **if** |points in the window| > threshold **then**
5: next x = mean of the x coordinate of these points
6: **else**
7: next x = current x.
8: **end if**
9: increment y by $\frac{h}{\#windows}$
10: **if** y haven't reach the top of the image **then**
11: go back to step 3
12: **else**
13: returns all the points captured by the windows.
14: **end if**

In our implementation, we use margin=50, #windows=20, and threshold=50

The result is shown as Figure 12. In our scenario, we want to find 4 lanes, so we find the peak of histogram in each quarter of the image and let the algorithm deal with 4 independent windows in each iteration.

3.2.5 Curve Fitting

For the captured edge points returned by sliding window, we first transform them back to the original coordinate, then fit a polynomial of degree=1 as in Method 1. The final result is shown as Figure 13.

3.3 Discussion

In this project, we have mentioned two traditional methods, here we roughly discuss our observation.

1. In traditional methods, we need to manually adjust parameters in order to achieve reasonable performance. For example, we need to adjust ROI regions and thresholds for both Hough

line detection and sliding windows for each different dataset. The parameters depends on the illumination of the scene, camera perspective, and road condition. See examples in Table 2.

2. In method 2, our reference method Automatic Addison does edge detection before perspective transform. After experiments we decide to do Canny edge detection after perspective transform. See the following comparison in Table 3. Notice that we get more edges with higher quality near the horizon with the right approach. This insight boot our accuracy on Tusimple from 60.3% to 82.1%

3. As mentioned in the previous point, we decide to apply Canny Edge detection after perspective transform. However, perspective transform creates two empty triangle regions at the bottom corners of the warped image, so the edge detector and sliding window would pick up these unwanted edges of the triangle, causing the bottom portion of fitting polynomial to deviate from the ground truth.(see the left column of Table 4)

   The solution we found for this issue is to concatenate on both left and right the horizontally flipped version of image before warping. We can thus fill the empty triangle regions and eliminate the unwanted edges.(As shown in the right column of Table 4)

   This modification further improve our accuracy from 82.1% to 85.7%.

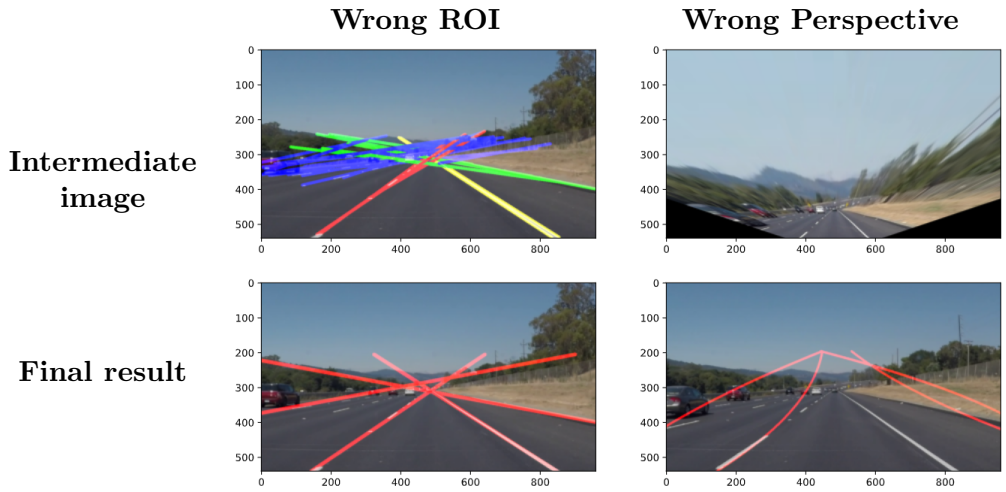


Table 2: If We applying the “ROI in Method1” and ”perspective transform in Method2 ” tuned for Tusimple on other road image, the result would be pretty shitty.

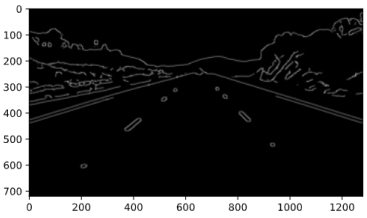
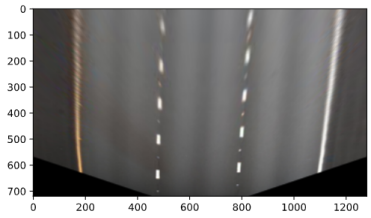
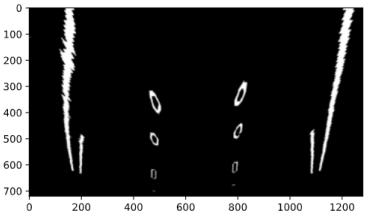
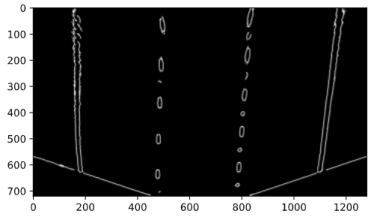
	Canny → Warp	**Warp → Canny**
Step 1		
Step 2		

Table 3: We find out that doing Canny edge detection after perspective transform gives more accurate lane lines.

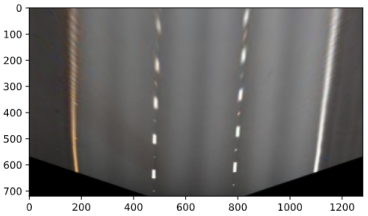
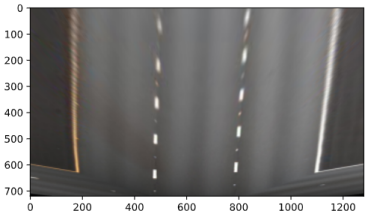
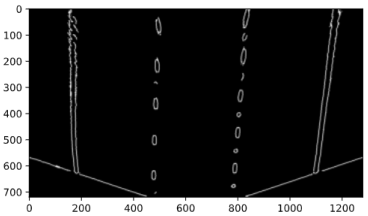
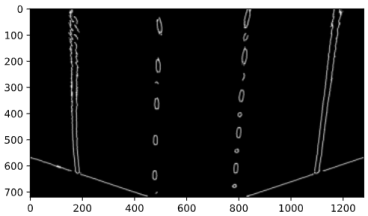
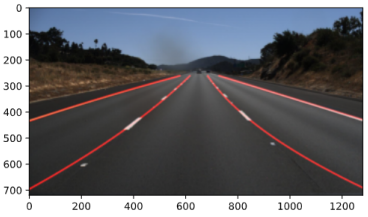
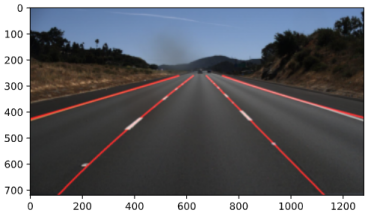
	Warp original image (Before Modification)	**Warp circular repeated image ((After Modification))**
After warping		
After Canny edge		
Final result		

Table 4: The left column shows the result before our modification and final output lines are a little skewed. The right column shows the result after modification and final output lines get straight.

4 Deep learning method

4.1 The network

For training, we set our batch size to 8 since the model is quite big. Which image to select, we use a random system. First, we randomly generate the number between [0,1] and choose an image from a specific lane number.

Before sending the data into the model for training, we will need to set the environment right, each data put in the correct place because we are using double hinge loss. First, we need to extract the ground truth coordinates from the JSON file. Since we have resized the images to the same size, the coordinates need to shift according to the resize ratio (we set resize ratio to 8). Then we separate each image into three categories:

1. Same instance (same lane).
2. Different instance but same class (different lane but close points).
3. Different instance and different class.

We introduce our network structure here. We cascade two hourglass networks as our model. Hourglass networks are a convolutional encoder-decoded network (meaning it uses convolutional layers to break down and reconstruct inputs). They take an image, and they extract features from this image by deconstructing the image into a feature matrix. Combining the feature matrix with early layers in the network with higher spatial understanding allows us to understand a lot about (what it is + where it is); in our case, it will know more about the lanes. Residuals are used heavily throughout the network. They are used to combine the spatial info with the feature info, and not only that, we use a particular type of residual known as bottleneck blocks. Bottlenecks, instead of having 3×3 convolutions, we have one 1×1 convolution, one 3×3 convolution, and one 1×1 convolution. This makes the calculations a lot easier on the computer (3×3 convolutions are much more intricate to do at scale than 1×1 convolutions), which means we get to save lots of memory. Figure 14 shows the structure of one hourglass network.

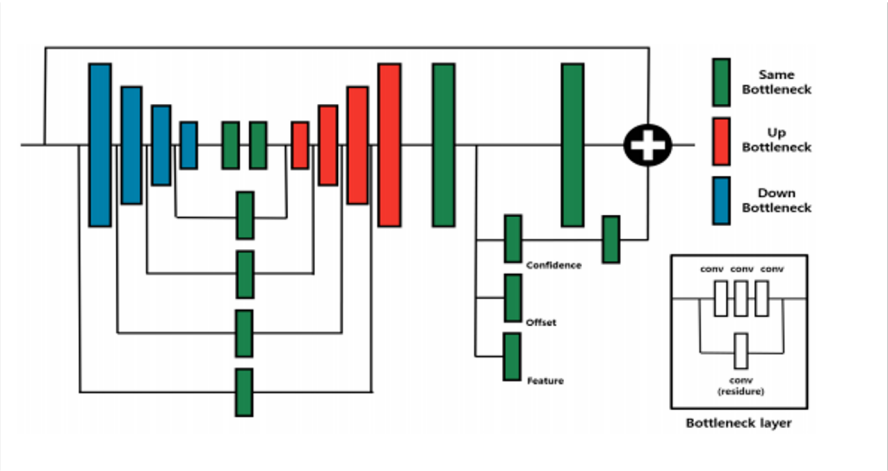


Figure 14: Hourglass network structure

The outputs of the network, which are confidence score, offset, and feature. They are corresponding to confidence map, similarity matrix, and semantic segmentation map in SGPN. First, we show the

confidence loss. Confidence loss consists of two parts, existence loss and non-existence loss. The existence loss is applied to the cells that include key points; the non-existence loss is utilized to reduce the confidence value of each background cell. The non-existence loss is computed at cells that predict confidence value higher than 0.01. Because cells away from key points converge rapidly, this technique helps the training concentrate on cells closer to the key points. The following shows the loss function of the confidence branch:

$$L_{exist} = \frac{1}{N_e} \sum_{c_c \in G_e} (c_c^* - c_c)^2 \tag{1}$$

$$L_{non\_exist} = \frac{1}{N_n} \sum_{\substack{c_c \in G_n \\ c_c > 0.01}} (c_c * -c_c)^2 + 0.00001 \cdot \sum_{c_c \in G_n} c_c^2 \tag{2}$$

Where N_e denotes the number of cells that include key points, N_n denotes the number of cells that do not have any key points, G_e denotes a set of cells that consist of key points, G_n denotes a set of cells that consist of points, c_c denotes the predicted value of each cell in the confidence output branch, c_c^* denotes the ground-truth value. Then we show the offset loss, which is the loss function for the offset branch. Offset loss is the predicted lane cell loss.

$$L_{offset} = \frac{1}{N_e} \sum_{c_x \in G_e} (c_x^* - c_x)^2 + \frac{1}{N_e} \sum_{c_y \in G_e} (c_y^* - c_y)^2 \tag{3}$$

Lastly, the embedding feature loss which is the loss function for the feature branch. SGPN inspires the loss function for this branch; a 3D points cloud instance segmentation method [2]. The branch is trained to make the embedding feature of each cell closer if the embedding features are the same in this instance. Equation (4) and (5) show the loss function of the feature branch:

$$L_{feature} = \frac{1}{N_e^2} \sum_{i}^{N_e} \sum_{j}^{N_e} l(i,j) \tag{4}$$

$$(i,j) = \begin{cases} \|F_i - F_j\|_2 & \text{if } I_{ij} = 1 \\ \max\left(0, K - \|F_i - F_j\|_2\right) & \text{if } I_{ij} = 0 \end{cases} \tag{5}$$

Where F_i denotes the predicted embedding feature of the cell i, I_{ij} indicates whether cell i and cell j are the same instance or not, and K is the constant such that $K > 0$. If $I_{i,j} = 1$, the cells are the same instance, and if $I_{i,j} = 0$, these cells are different instances. When the network is trained, the loss function makes features closer when each cell belongs to the same instance; it distributes features when cells belong to different instances. The ultimate loss function is as follow:

$$L_{total} = \gamma_e L_{exist} + \gamma_n L_{non\_exist} + \gamma_o L_{offset} + \gamma_f L_{feature} \tag{6}$$

For optional, we also add OLE loss [1] into our loss function, which is a modified version of the geometric loss. Again, we won't go deep into this because it is an optional feature; this does not make the result better. If the reader wants to know more, please refer to the paper [1].

4.2 Postprocessing

Postprocessing is for drawing out the lanes. The steps are as follow:

1. Find six starting points. Starting points are defined as the three lowest points and the three leftmost or rightmost points. If the predicted lane is on the left related to the center of the image, the leftmost point is selected.
2. Select three closest points to the starting point among points that are higher than each starting point.
3. Consider a straight line connecting two points that are selected at steps 1 and 2.
4. Calculate the distance between the straight line and other points.
5. Count the number of points that are within the margin. The margin γ is set to 1.
6. Select the point with a maximum and larger count than the threshold as a new starting point, and consider that the point to the same cluster with a starting point. Then, we set the threshold to twenty percent of the remaining point.
7. Repeat from step 2 to step 6 until no points are found at step 2.
8. Repeat from step 1 to step 7 for all starting points, and consider the maximum length cluster; as a result lane.
9. Repeat from step 1 to step 8 for all predicted lanes.

After the postprocessing, our model predicts n lanes for each image, and every lane will be marked with the same color.

4.3 Experiment Detail

We train for 1000 epochs, and there are plenty of interactions in one epoch depend on the batch size (we set the batch size to 8). We use Adam as our optimizer and set the learning rate to 0.0001. We set $\gamma_e, \gamma_n, \gamma_o, \gamma_f$ all to 1.

4.4 Discussion

Although this method shows high accuracy in the TuSimple dataset, it is not robust enough to apply on the road. The reasons are: (1) It does not perform well in dark and bad weather like rainy days. (2) The camera must be at a specific angle to shoot for this method to work because we trained on the TuSimple dataset, and this dataset was shot in the same way. (3) If the lane has multiple cross lines, our network will detect the cross lines instead of the actual lane. (4) Our network performs poorly in intersections. We have found the reason for the disadvantages we also have the solution. Because the TuSimple dataset doesn't include the cases we mention above, so our network does not learn to cope with these scenarios. The solution is to feed more different kinds of data into our network. CULane seems like an ideal dataset, but due to the time constraint, we didn't try it.

5 Experimental Results

In this section, we demonstrate the results of three methods respectively. Later, we put three methods together for comparison.

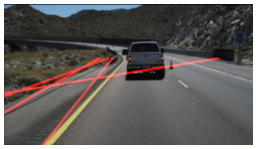
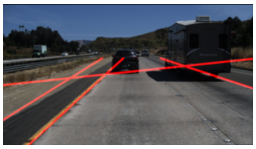
Good				
Acceptable				
Bad				

Table 5: Resulting images for Method 1

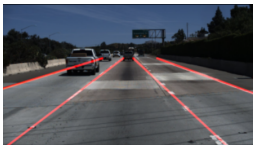
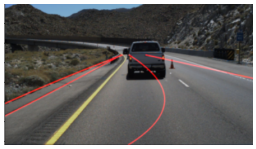
Good				
Acceptable				
Bad				

Table 6: Resulting images for Method 2

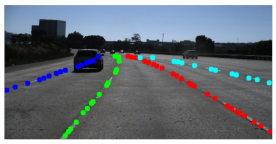
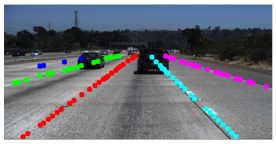
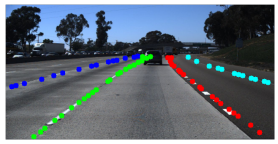
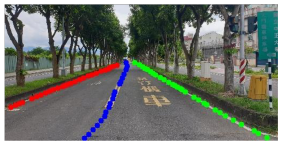
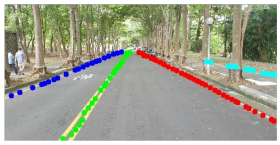
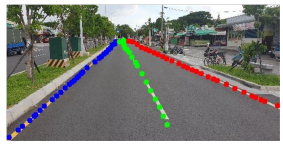
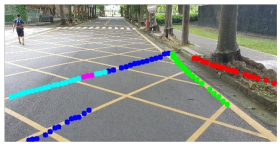
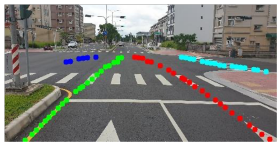
TuSimple				
My data				
Fail cases				

Table 7: Resulting images for deep learning method

From Table 5, 6, 7, we can see that Method 1 and 2 can draw correct lanes when the path is straight while have bad results often when there are cars or curve on the road. However, deep learning method can conquer this problem. The main issue for deep learning method is that multiple cross lines confuse the model to distinguish the correct lanes, while this can be fix by training on dataset with this kind of data.

5.1 Comparison

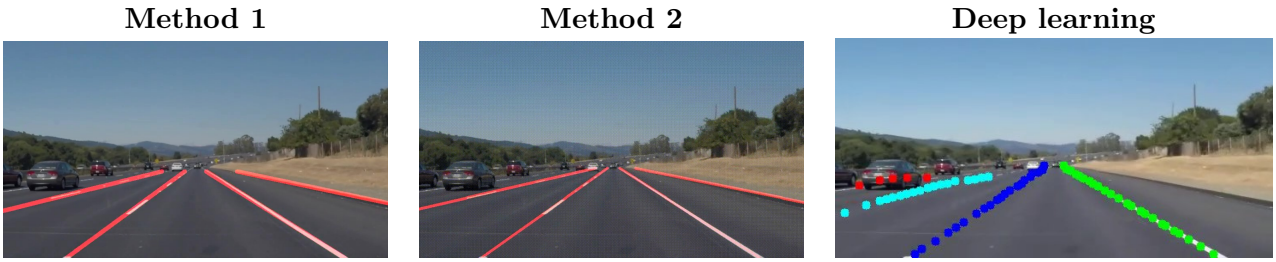


Table 8: Comparison under same scenario

Table 9 shows that three methods can all achieve nice accuracy. While deep learning method perform much better than both traditional methods.

Traditional method	**Loss**	**Accuracy**	**FP**	**FN**
Method 1	-	0.73	0.57	0.48
Method 2	-	0.82	0.47	0.35
Method 2 (use repeated image in perspective transform)	-	0.86	0.40	0.27
Deep learning method	**Loss**	**Accuracy**	**FP**	**FN**
w/o OLE Loss	1.11	0.97	0.02	0.14
w OLE loss	1.57	0.94	0.08	0.20

Table 9: Comparison the numerical results between three methods

6 Conclusion

Overall, We can see that traditional methods have more constraints in contrast to deep learning method. With obstacles on the road or curve on the way all lead traditional method to fail, yet deep learning method has much stable performance.

In this project, we show three different methods on lane detection task. First, we implement them and all of them can reach accuracy to over 70%. Second, we successfully increase the accuracy on Method 2 by modify the pipeline with what we learn form class. Third, we tried the latest method on deep learning model also compare it with two different loss. Although there are still some issues have to conquer for all of them, we have tried our best on this task.

7 Work assignment

Team member:

資科工碩: 309551178 秦紫頤
電資學士班: 0610021 鄭伯俞
清華: H092505 翁紹育

Traditional methods:

鄭伯俞、翁紹育

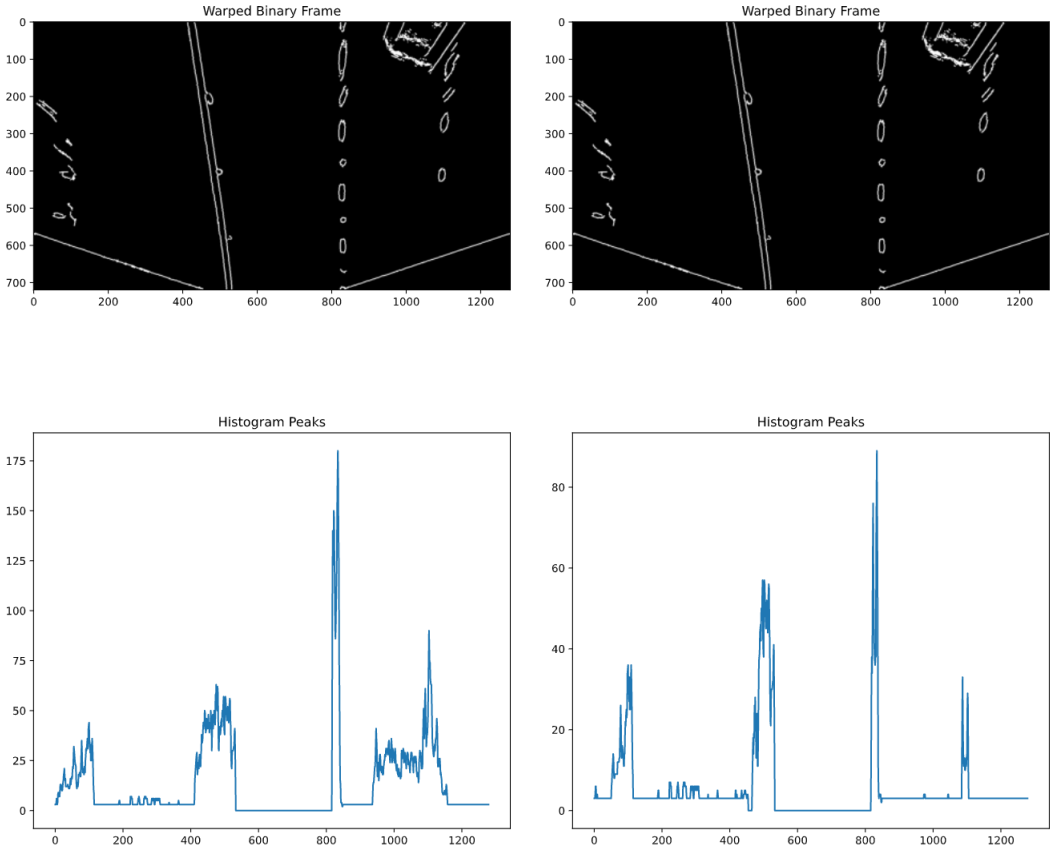
Deep learning method:

秦紫頤

References

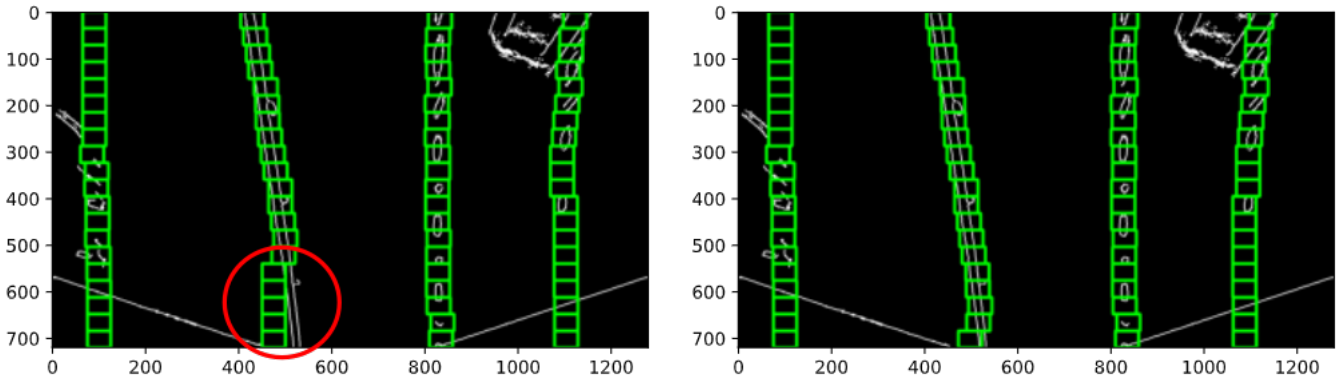
[1] W. Van Gansbeke, B. De Brabandere, D. Neven, M. Proesmans, and L. Van Gool. End-to-end lane detection through differentiable least-squares fitting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[2] W. Wang, R. Yu, Q. Huang, and U. Neumann. Sgpn: Similarity group proposal network for 3d point cloud instance segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2569–2578, 2018.



(a) Histogram with whole image

(b) Histogram with buttom half image



(c) Sliding Window Result of (a)

(d) Sliding Window Result of (b)

Figure 11: Histogram with buttom half image performs better

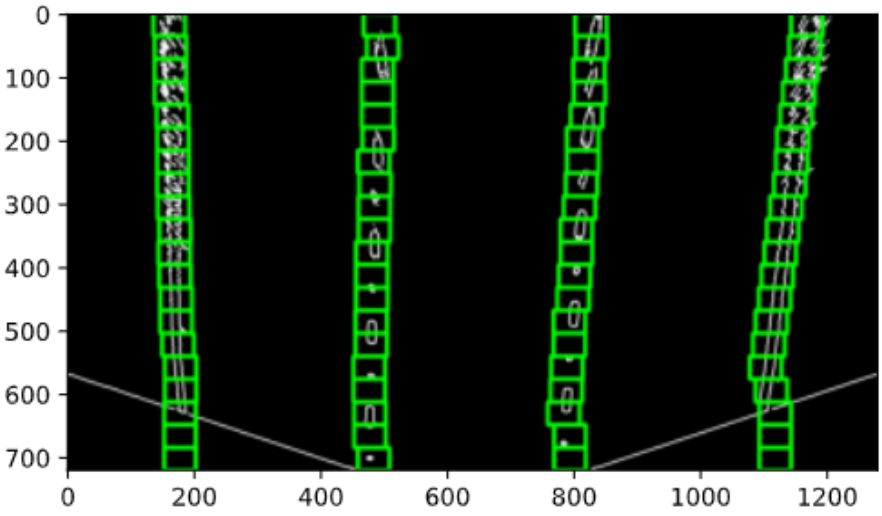


Figure 12: Sliding window under our parameters setting

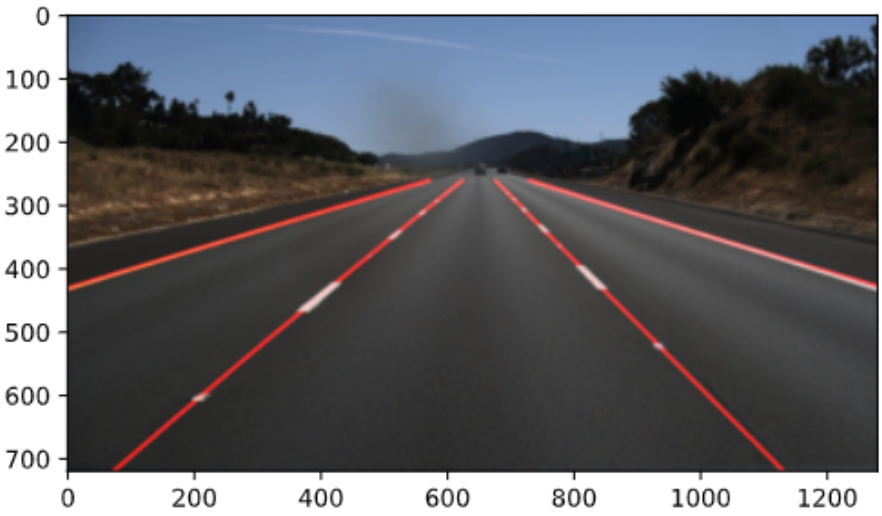


Figure 13: Result of method 2