Lane Detection: CS221 Final Report

Zhiling Huang, Xinyi Yang, Tao Jia

1. Introduction

Lanes on the road are designated to use by a single line of vehicles. Lane are separated by painted white or yellow (double solid, solid, or dotted) lines. In autonomous driving, lane detection provides reference information to the control of a car. The purpose of this project is to detect the location of lane separation lines ('lane' thereafter) on the video clip (represented by a set of frames) and whether a lane change happens.

Our data comes from a lane detection challenge hosted by tu Simple Benchmark[1]. The setting is car driving on highways. The video clips have the following characteristics:

- The view direction of the camera is very close to the driving direction.
- Good or medium weather conditions.
- Different daytime.
- The number of lanes are 2, 3, 4, or more, but only the central 4 lanes (from driver's view) are required.

The dataset contains 7000 one-second-long video clips of 20 frames each. The last frame contains labeled lanes, which are marked by polylines. The training set has 3626 video clips with the last frames annotated. The test set has 2782 video clips without annotated frames. For every last frames in the training set, there is a label in the format of a list of array. Each array stores information about a (potential) lane. For each fixed height in the image (y in range (240, 720, 10)), the label indicates the corresponding x-axis value if a lane appears at this height, and -2 if the lane does not appear at this height. The label is stored in JSON format. For example, the following label indicates a lane that ranges across (632, 280) to (299, 710) on an image.

'[-2, -2, -2, -2, 632, 625, 617, 609, 601, 594, 586, 578, 570, 563, 555, 547, 539, 532, 524, 516, 508, 501, 493, 485, 477, 469, 462, 454, 446, 438, 431, 423, 415, 407, 400, 392, 384, 376, 369, 361, 353, 345, 338, 330, 322, 314, 307, 299]'

Our prediction of the lane will also be output in this format to remain consistency. Details about the construction of the evaluation metrics can be found on Section 3.

[1] http://benchmark.tusimple.ai

2. Literature Review

1) Yolo is a real-time object detection method that applies a single neural network to the whole image. The network separates the image into regions and predicts for each region bounding boxes and a probability for each box that lane exists in it. The model makes predictions with a single network evaluation, which makes it extremely fast.

Yolo is the first method we tried for lane detection. We set up Google Cloud and used two Tesla K80 to train the model, but after days of training, we conclude that the method is not most appropriate for lane detection.

Due to the limitation of Yolo's algorithm, it needs to be trained with image and bounding boxes with labels. Its output format is also boxes, with a probability for each box, which is the probability that a lane exists in the bounding box. But boxes, which are rectangles, are not appropriate shapes to describe where a lane is, which is more like a line.

Even tuning the bounding box size won't address this limitation. We tried using a large bounding box to describe each lane, but that doesn't work because $> 95\%$ of the information in the bounding box will not be lane (lane is only a slim line); We also tried using dozens of small bounding boxes to describe one lane, which also does not work because Yolo cannot recognize a series of bounding boxes as one object. Therefore Yolo is not ideal for lane detection.

2) Udacity has a project for lane detection. It provides little guidance for us. We want to predict 4 lanes (if they exist) and Udacity can only come up with the nearest two lanes; Udacity's lane detection project cannot observe lane change because it only considers one image.

3. Evaluation

We use two different metrics to evaluate the quality of lane detection.

The first metric is the sum of squared error for all the required points in the image. Total error $= \sum(X_{pred} - X_{true})^2$ for all required points in the label. The smaller the total error, the better the prediction is.

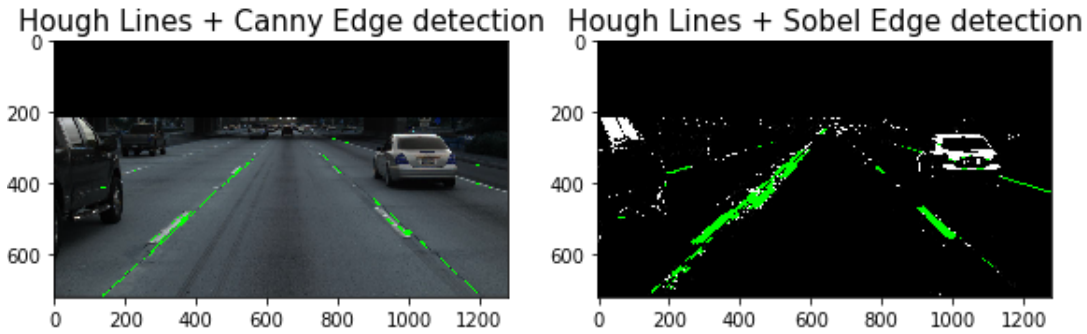
The second metric is the ratio of the correct predicted points, or total number of prediction within acceptable error tolerance divided by the total number of required points.

In math format, the ration of the correct predicted points is:

$\frac{\text{Number of predicted points within error tolerance}}{\text{Number of required points}}$. We use multiple error ranges to check for robustness, including 5px, 10 px, 15px, and 20px. This metric is between 0 and 1. The larger the metric, the better the prediction is.

4. Baseline & Oracle

Our baseline approach is to use canny edge detection to detect the line. Below is a picture showing the comparison of the edge detected using the baseline method and our improved method (details in the next section):



The oracle is the lane detected by human beings. We estimate the accuracy resulted from human judgment will be nearly 100%.

5. Our Approach

i) **Overview**
The whole system is structured as following:

ii) **Improved Single Image Lane Detection**
We did a pre-processing in order to better detect the lane on the image.

1. At the very first step, we cropped the upper part of the image (y=0 160), because it usually includes trees/buildings/clouds that have very clear edges and does not contribute to lane detection. If we include this part, the edges of these items will lead to bias in lane detection.

2. We did a combination of Sobel transformation and Black-and-white binarization. We set the threshold of binarization to be twice the threshold of using Otsu's method, and set max threshold to be 0.8 to prevent overflow:

$$thres = \max(2 \times thres_{ostu}, 0.8) \tag{1}$$

This autothreshold method can eliminate the difference between different weather conditions at which the photo was taken. It works for most of pictures, but there are still 10% pictures where lanes failed to be identified as white. So we added Sobel edge detection, which can find the edges of lane marks, as a complement to the autothreshold procedure.

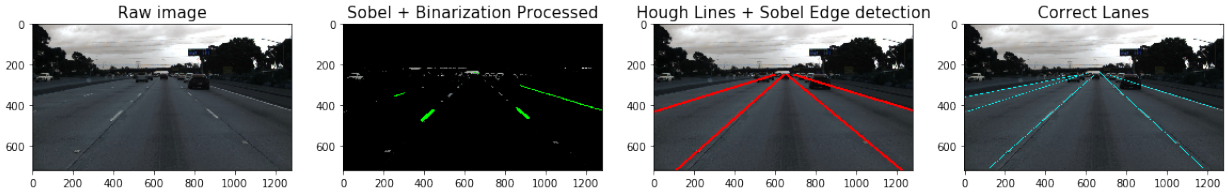
3. We did a probabilistic Hough line transformation on the preprocessed images. It is a transformation to detect straight lines, and should be applied after an edge detection pre-processing. Lines can be expressed in polar system: $y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$. For each point, we can plot a family of lines going through it, and draw a set of curves for all points in the image where the x-axis represents θ and the y-axis represents r. If two curves intersects, it means they belong to the same line with the coordinates (θ, r) as its parameters. The more curves going through an intersection, the line represented by that intersection have more points. If the number of points above a threshold, we can use the parameters of an intersection (θ, r) to indicate a line. We used OpenCV function HoughLinesP to implement the transformation and set the parameters as following:

    - r: The resolution of the parameter r in pixels. We used 1 pixel.
    - θ: The resolution of the parameter theta in radians. We used 1 degree (np.pi/180).
    - threshold: The minimum number of intersections to detect a line. We set it to 100.
    - minLinLength: The minimum number of points that can form a line. We used 150.
    - maxLineGap: The maximum gap between two points to be considered in the same line. We used 10.

4. Probabilistic Hough line transformation output the two extremes of each detected line in the format of (x1, y1, x2, y2). We calculated the corresponding slope for each detected line, and used sorting and K-means to cluster the Hough lines into 2-5 groups indicated 2-5 different lanes. We then aggregated Hough lines in each group to find the slope and intercept of each line, and output the image with the prediction lines as well as the prediction in the form of arrays (the list of x-value for y = 240, ..., 710 for each lane).

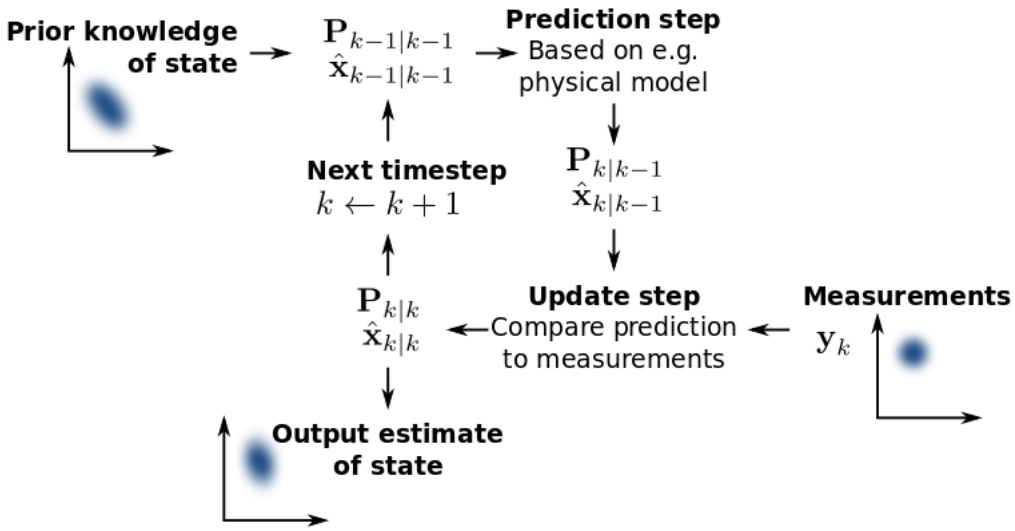
The following pictures show the whole process of single image detection. The rightmost

picture illustrates the correctly labeled lanes.



iii) **Use Multiple Images to Make More Accurate Prediction**
While in ideal cases (view not blocked by other cars, lane marks clear and continuous) we can obtain good results from Single Image Lane Detection, this simplistic algorithm is challenged by real-world situations. Thankfully, our dataset consists of one-second video clips, so we can overcome some of the problems in the last picture by tracing the previous ones in the same clip.

We use Kalman filter for this approach. Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each time frame. An intuitive tutorial of Kalman filter can be found in [5]. A one-line description of Kalman filter: it is the Hidden Markov Model in the world of continuum. Therefore it provides an ideal method for lane detection, where the lanes are best described as lines/polynomials on the picture.

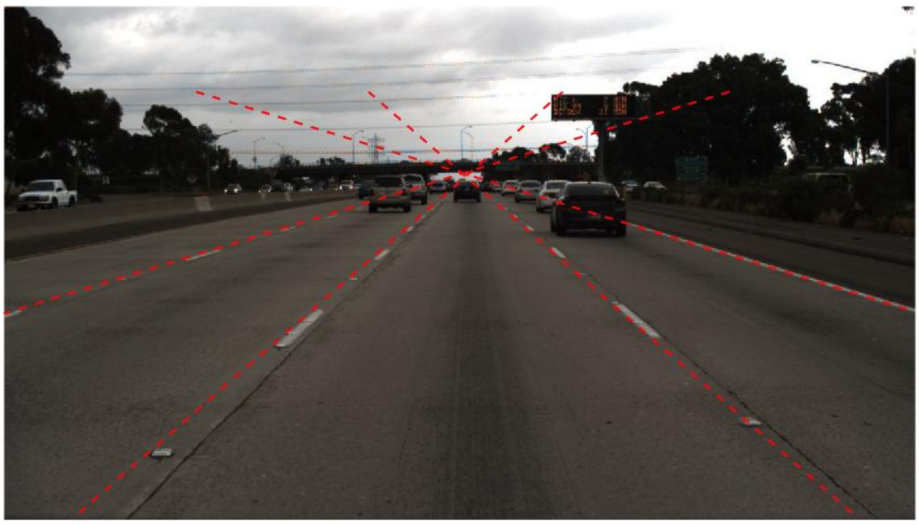


After the preprocessing steps mentioned above (autothreshold binarization and edge detection) and Hough Line Detection, we find two local maxima near the center of image and

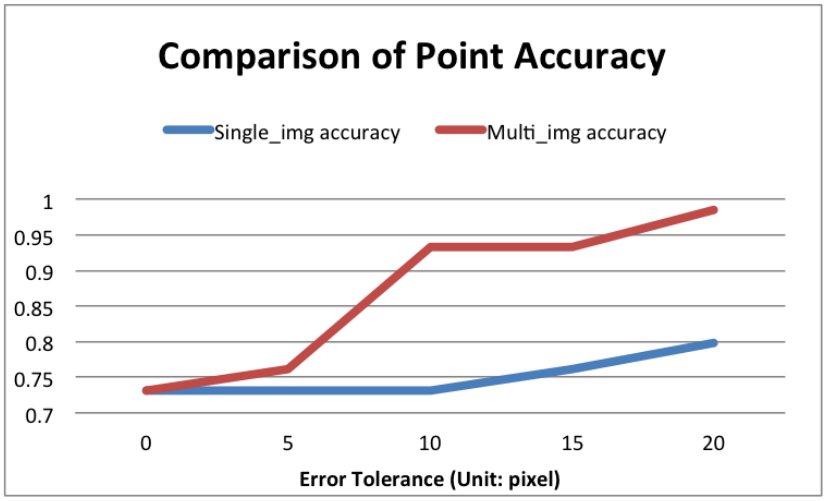
output them to Kalman filter. In the Kalman filter part, we save the previously-detected lanes in a repository, and count the number of times each lane has been detected. Then we match the lanes found in the current picture with those in the repository. If a current lane is similar enough to another lane in the repository, the repository will be updated with the lanes' current location; if a current lane does not match any previous lanes, it will be abandoned and we will look for the third local maxima. We used MATLAB and Simulink to apply this algorithm.

With this method we first do detection on the two most distinct lanes (most likely the two central lanes); naively applying this method to all the four lanes in our video clips gave poor results. However, our goal is to detect all the lanes. We found a simple algorithm helpful: we first detect the two central lanes, then "black out" the region between the two lanes by changing the pixels in this region to black, so that the system will be sensitive enough to find the next two local maxima (the left most lane and the right most lane). After this step we can find all the four lanes.

For example, for the following graph, we first detected the two most distinct red lines in the middle. And then "blacked out" the triangular area formed by these two lines and the bottom edge of the picture. Now the system is sensitive enough to detect the left most red line and the right most red line. At the end, we merge these lines together into one picture.



We compare our results using only 20th frame (method ii) and the results using Kalman filter with multiple images. With Kalman filter, predictions become more robust. $> 95\%$ of the predicted points are within 20 pixels of ground truth and 73% of the predicted points overlap with the true labels. The average accuracy rate is shown below:

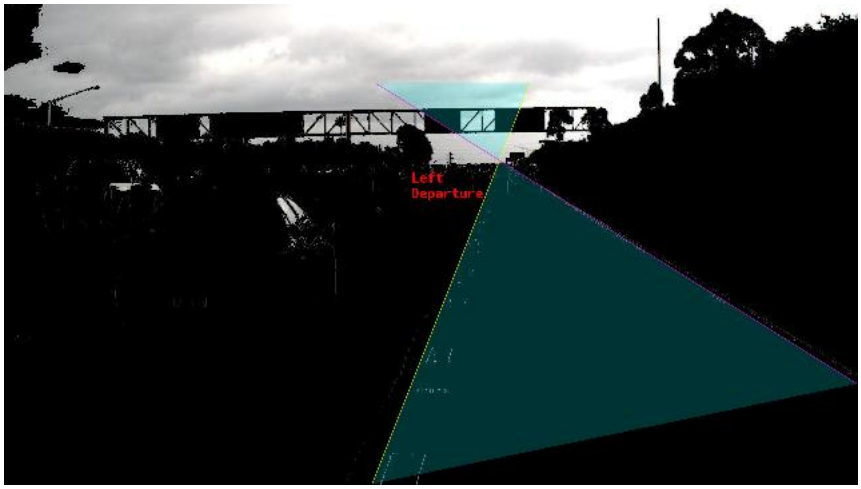


iv) **Lane Change Detection**
The detection of lane change and lane departure is a key application of lane detection: if the driver gets drowsy and deviates from the center of lane, a loud alarm can get him back from danger.

The Lane Change Detection part uses the Hough Lines block, to convert the Polar coordinates of a line to Cartesian coordinates. The subsystem uses these Cartesian coordinates to calculate the distance between the lane markers and the center of the video bottom bound-

ary. If this distance is less than the threshold value, the example issues a warning.

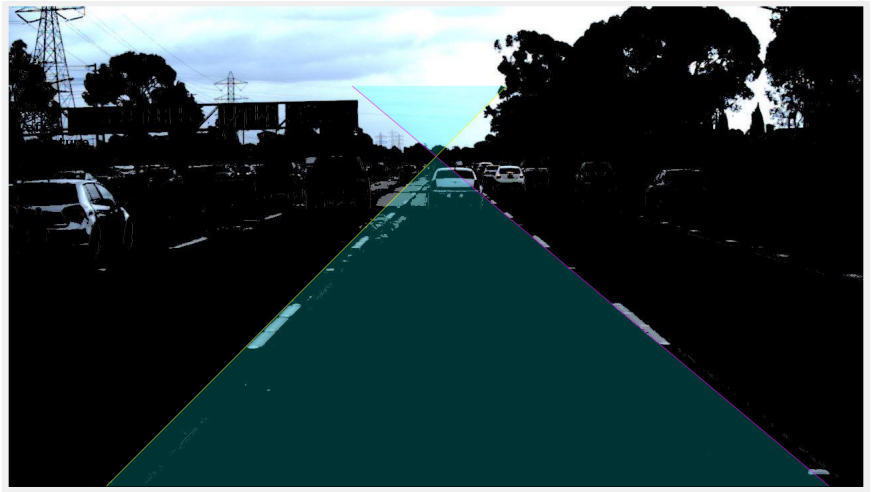
Take the following picture as an example: the system detects that the left lane (denoted with a yellow line) is too close to the center of the image. The system predicts that the driver is trying to change to the left lane. Therefore it reports "Left Departure".



6. Error analysis

i) Painted White Lines not Obvious
Painted white lines that are closer to the camera are usually clear and the predicted lanes are accurate. But painted white lines that are further sometimes will appear to be vague.

We see slight errors in the prediction of the following graph:



The purple predicted line, which is the line to the right, adheres closely to the correct lane, when it is both close and far. However the yellow line, though accurately describing the

lane when it is close, is a little off (to the left of the true painted white line) when the lane extends further.

More specifically the yellow line passes through an area with a lot of white points (to the left of the car). The preprocessing should have removed most of them, and only preserved the points related to the painted white lines. However since the white lines are not clear, the preprocessing step preserved too many white points and eventually confused the algorithm that predicts lane.

If we take a look at the original picture, the reason becomes clear:

In the yellow box, or to the left of the car, the ground is brighter, making the painted white lines less obvious; while in the purple box, or to the right of the car, the ground is darker and making the painted white lines more obvious. As a result, preprocessing is more successful with the lane to the right. For the left lane, preprocessing will leave more noise and deviate the prediction.

ii) Other Causes of Error
We noticed some edge cases. For example, defunct white lines that are not completely erased confuses the system; images taken when the camera is not facing the correct direction causes trouble; etc. These edge cases only happen rarely, so they don't have a huge impact on our prediction accuracy. However, if we want to use lane detection in real life, these edge cases have to be tackled and are places that we can improve.

7. Conclusion

We have realized a method that can detect multiple lanes in video clips of road images. We use Hough line detection for single-image lane detection, and use Kalman filter for multi-image lane detection and lane tracking. Our method also enables detection of lane change and deviation. This algorithm is robust on different weather conditions and traffic conditions. The current performance is limited partly because the data was given as 1 second video clips;

longer videos should see a better result from our method.

A natural improvement on this work would be to use polynomial functions in Hough line detection, so that we can detect curvy roads better. Also our detection speed (excluding preprocessing) is limited to around 6 frames per second, not yet reaching the speed of most videos. If we combine detection with preprocessing, the total time to process one video may become intolerable. Possible ways to accelerate this process includes using GPU for preprocessing and compressing images before preprocessing.

8. Moxel

Last but not least, we also uploaded our model to moxel:
http://beta.moxel.ai/models/albert/lanedetection/latest. Since moxel only supports single image input, we are only demoing single image lane detection. Try it out!

References

[1] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, *You Only Look Once: Unified, Real-Time Object Detection* , arXiv:1506.02640v5 [cs.CV]

[2] Joseph Redmon, Ali Farhadi, *YOLO9000: Better, Faster, Stronger*, arXiv:1612.08242v1 [cs.CV]

[3] AlexeyAB/darknet (a forked version of original darknet/YOLO, with detailed discussion on training without using extra image sources):

`github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects`

[4] *Documentation of Hough line transformation in OpenCV*,

`https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.ht`

[5] *How a Kalman filter works, in pictures*,

`http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/`