UDACITY

< Back to Self-Driving Car Engineer

Advanced Lane Finding

REVIEW

CODE REVIEW

HISTORY

Requires Changes

1 SPECIFICATION REQUIRES CHANGES

You have done a big job and have a very good first step! It is a really tough project. And it is awesome that you almost complete it 👍
Please correct a few small issues mentioned below to meet specification and you are done! Good luck!

Writeup / README

The writeup / README should include a statement and supporting figures / images that explain how each rubric item was addressed, and specifically where in the code each step was handled.

Well done with writeup! 👍
It is always important to understand advantages and limitations of your algorithm and know ways to improve it.

Camera Calibration

OpenCV functions or other methods were used to calculate the correct camera matrix and distortion coefficients using the calibration chessboard images provided in the repository (note these are 9x6 chessboard images, unlike the 8x6 images used in the lesson). The distortion matrix should be used to

un-distort one of the calibration images provided as a demonstration that the calibration is correct. Example of undistorted calibration image is Included in the writeup (or saved to a folder).

Well done with camera calibration process! Here is more info about this process in documentation if you are interested:
http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html?
http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html
http://docs.opencv.org/3.1.0/dc/dbb/tutorial_py_calibration.html

And very detailed Microsoft research:
http://research.microsoft.com/en-us/um/people/zhang/calib/

Pipeline (test images)

Distortion correction that was calculated via camera calibration has been correctly applied to each image. An example of a distortion corrected image should be included in the writeup (or saved to a folder) and submitted with the project.

`undistort` function is successfully used for images.

Remember that usually Python is used by self-driving car engineers for prototyping and final algorithms are usually implemented in C/C++ as it is faster than Python and almost all data are handled in self-driving car in real time. Here you can find article about distortion correction in C++:
https://medium.com/@mimoralea/but-self-driving-car-engineers-dont-need-to-know-c-c-right-3230725a7542#.ge488ni33

Note please that you will apply C++ in term 2 (though in other context).

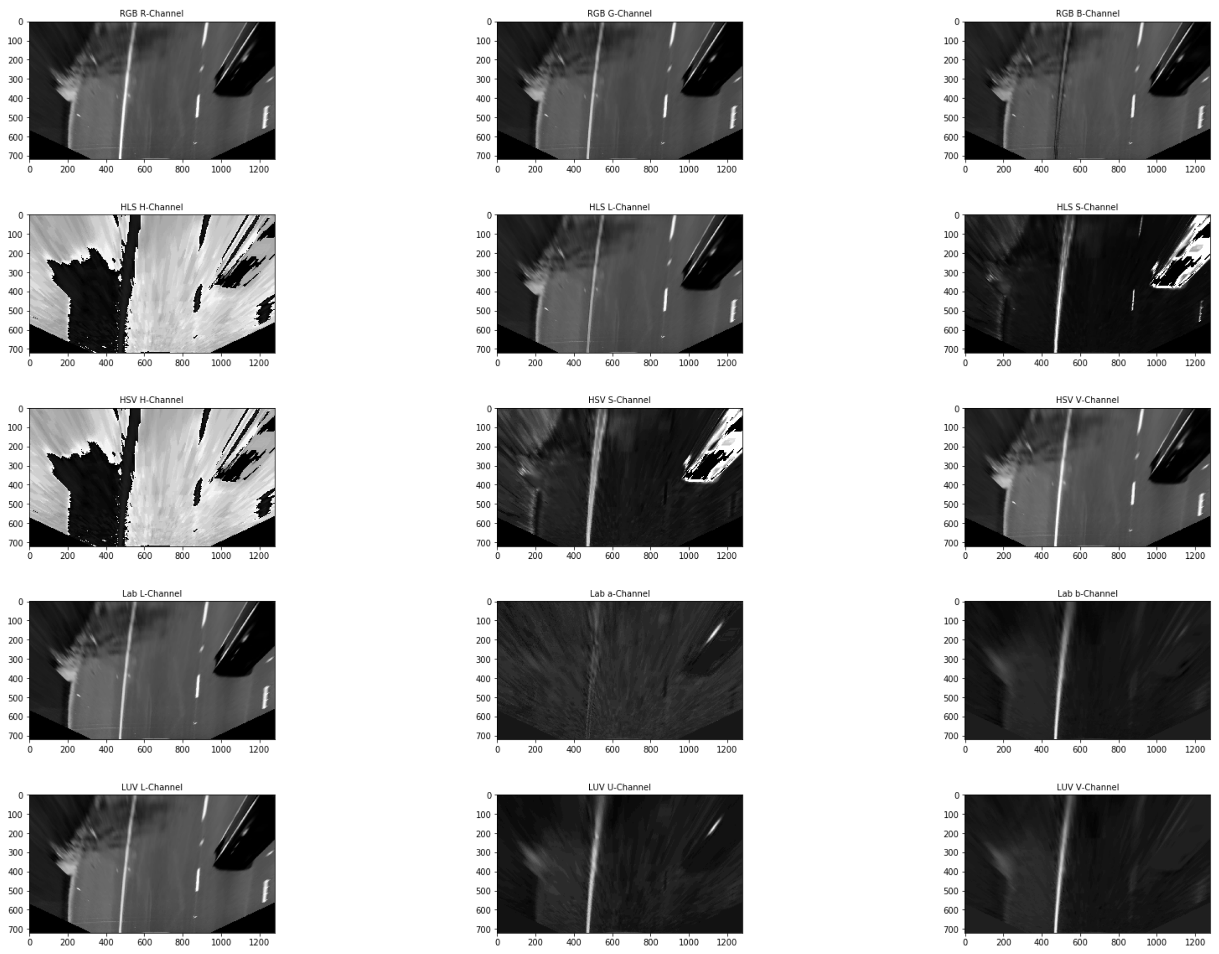
A method or combination of methods (i.e., color transforms, gradients) has been used to create a binary image containing likely lane pixels. There is no "ground truth" here, just visual verification that the pixels identified as part of the lane lines are, in fact, part of the lines. Example binary images should be included in the writeup (or saved to a folder) and submitted with the project.

Well done with color transforms and gradient methods to create binary images for further lane detection!

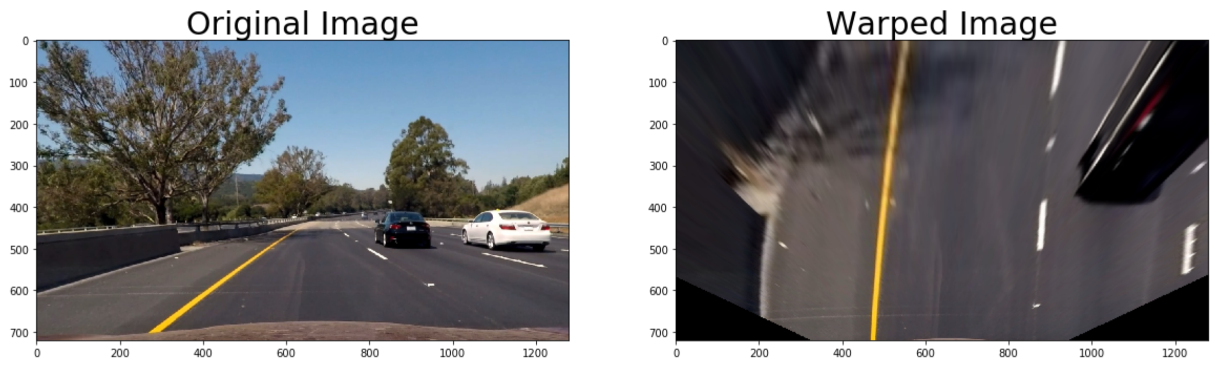
Briefly about lines, colorspaces, and channels in different colorspaces:

- S-channel of HLS colorspace is good to find the yellow line and in combination with gradients, it can give you a very good result.
- R-channel of RGB colorspace is pretty good to find required lines in some conditions.
- L-channel of LUV colorspace for the white line.
- B-channel of LAB colorspace may be good for the yellow line.
  You just need to choose good thresholds for them.

Here are some examples of mentioned channels (in birds-eye view for the original image provided below):



Original image for channels shown above:



Note please also that gradients can give big noise in some conditions (for example with shadowed road or when lightning conditions changes drastically).
If you use Sobel remember that kernel size is limited to 1, 3, 5, or 7 values only:
http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html#cv.Sobel

Here is a code to build a widget for parameters fine tuning (for jupyter notebook only):

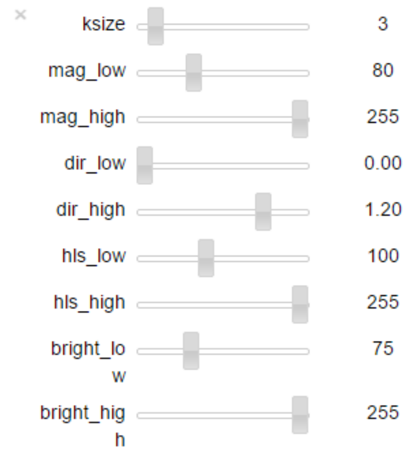
```
from IPython.html import widgets
from IPython.html.widgets import interact
from IPython.display import display

image = mpimg.imread('path_to_your_image')
image = cv2.undistort(image, mtx, dist, None, mtx)

def interactive_mask(ksize, mag_low, mag_high, dir_low, dir_high, hls_low,
 hls_high, bright_low, bright_high):
    combined = combined_binary_mask(image, ksize, mag_low, mag_high, dir_l
ow, dir_high,\
                                    hls_low, hls_high, bright_low, bright_
high)
    plt.figure(figsize=(10,10))
    plt.imshow(combined,cmap='gray')

interact(interactive_mask, ksize=(1,31,2), mag_low=(0,255), mag_high=(0,25
5),\
         dir_low=(0, np.pi/2), dir_high=(0, np.pi/2), hls_low=(0,255),\
         hls_high=(0,255), bright_low=(0,255), bright_high=(0,255))
```

Where `combined_binary_mask` function returns binary combination of different color and/or gradient thresholds and

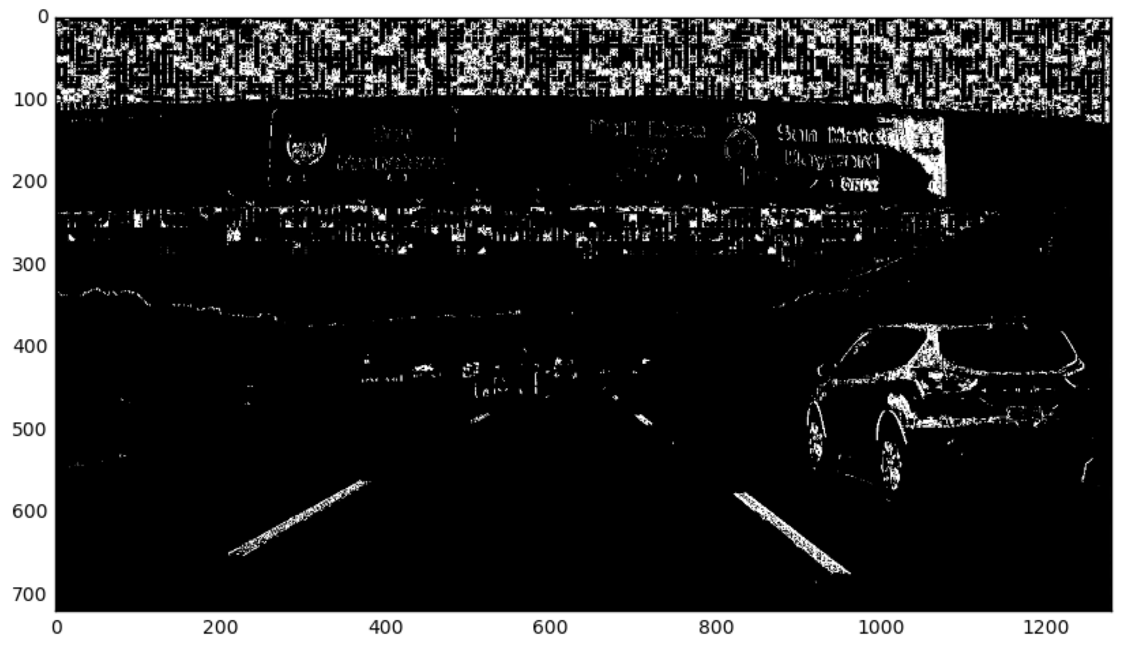
```
ksize,
mag_low,
mag_high,
dir_low,
dir_high,
hls_low,
hls_high,
bright_low,
bright_high
```

are parameters for thresholding (if you use other, you can change this part of code).

As a result, you will receive something like that and will be able to fine-tune parameters on the fly:



```
ksize        3
mag_low      80
mag_high     255
dir_low      0.00
dir_high     1.20
hls_low      100
hls_high     255
bright_low   75
bright_high  255
```



and will be able to see an impact of each parameter on the resulting binary image with defined combination of thresholds.

Here is similar decision for plain python (without notebook):

```
def adjust_threshold():
    image1 = cv2.imread("./test_images/test1.jpg")
    image2 = cv2.imread("./test_images/test2.jpg")
    image3 = cv2.imread("./test_images/test3.jpg")
    image4 = cv2.imread("./test_images/test4.jpg")
    image5 = cv2.imread("./test_images/test5.jpg")
    image6 = cv2.imread("./test_images/test6.jpg")

    image = np.concatenate((np.concatenate((image1, image2), axis=0),np.co
ncatenate((image3, image4), axis=0)), axis=1)
    image = np.concatenate((np.concatenate((image5, image6), axis=0), imag
e), axis=1)

    threshold = [0 , 68]

    kernel = 9
```

```
direction = [0.7, 1.3]
direction_delta = 0.01

gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)


cv2.namedWindow('test', cv2.WINDOW_NORMAL)

cv2.waitKey(0)

while True:
    key = cv2.waitKey(1000)
    print("key = ", key)
    if key == 55: # key "Home"
        if threshold[0] > 0:
            threshold[0] = threshold[0] - 1
        if direction[0] > 0:
            direction[0] = direction[0] - direction_delta

    if key == 57: # key "PgUp"
        if threshold[0] < threshold[1]:
            threshold[0] = threshold[0] + 1

        if direction[0] < direction[1] - direction_delta:
            direction[0] = direction[0] + direction_delta

    if key == 2424832: # left arrow
        if threshold[1] > threshold[0]:
            threshold[1] = threshold[1] - 1

        if direction[1] > direction[0] + direction_delta:
            direction[1] = direction[1] - direction_delta

    if key == 2555904: # right arrow
        if threshold[1] < 255:
            threshold[1] = threshold[1] + 1

        if direction[1] < np.pi/2:
            direction[1] = direction[1] + direction_delta

    if key == 49: # key "End"
        if(kernel > 2):
            kernel = kernel - 2
    if key == 51: # key "PgDn"
        if(kernel < 31):
            kernel = kernel + 2
    if key == 27: # ESC
        break

    binary = np.zeros_like(image)
    binary[(gray >= threshold[0]) & (gray <= threshold[1])] = 1

    cv2.imshow("test", 255*binary)
    print(threshold)
    print(direction)
    print(kernel)
```

You will be able to tune different parameters using keypress. The script works in an infinite loop. Instead of this plain python, you can also use cv2 user interface API for this purpose: http://docs.opencv.org/2.4.11/modules/highgui/doc/user_interface.html#

And here is also a helper code to find yellow and white lines separately if you want to investigate this question further:

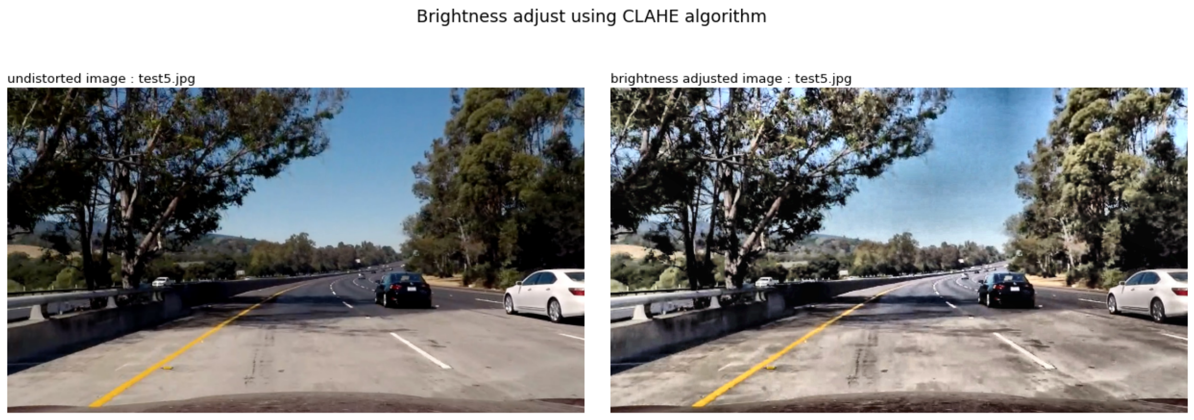
```
    b = np.zeros((img.shape[0],img.shape[1]))

    def thresh(img, thresh_min, thresh_max):
        ret = np.zeros_like(img)
        ret[(img >= thresh_min) & (img <= thresh_max)] = 1
        return ret

    hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
    H = hsv[:,:,0]
    S = hsv[:,:,1]
    V = hsv[:,:,2]

    R = img[:,:,0]
    G = img[:,:,1]
    B = img[:,:,2]

    t_yellow_H = thresh(H,10,30)
    t_yellow_S = thresh(S,50,255)
    t_yellow_V = thresh(V,150,255)

    t_white_R = thresh(R,225,255)
    t_white_V = thresh(V,230,255)

    b[(t_yellow_H==1) & (t_yellow_S==1) & (t_yellow_V==1)] = 1
    b[(t_white_R==1)|(t_white_V==1)] = 1
```

One more approach is to use a voting system for the final thresholding instead of the logical combination of different binary images. When the threshold is very clean like B channel from LAB space, it will have a relatively high confidence number. At the end, all the confidence number are summed up and compared with a total threshold. However, this method takes more run time than simply taking some combinations:

```
    conf_1 = 1
    conf_2 = 2
    conf_3 = 3
    threshold_vote = 3
    addup = img_sobelAbs*conf_1 +img_sobelMag*conf_1 +img_SThresh*conf_1 +
img_LThresh*conf_2 +img_sobelDir*conf_1 + img_BThresh*conf_3

    combined[(addup >= threshold_vote)]=1
```

Where `img_sobelAbs`, `img_sobelMag`, `img_SThresh`, `img_LThresh`, `img_sobelDir`, `img_BThresh` are different color or sobel thresholds.

Also you can notice that in some places of the video there are very intensive shadows. You can pre-processed the images through a brightness adjustment filter using a Contrast Limited Adaptive Histogram Equalization (CLAHE) algorithm. This will help you better detect the yellow lane line road markers that have over-cast shadows in the original image for subsequent steps (e.g. getting a binary threshold color mask for yellow and white lane lines).


Brightness adjust using CLAHE algorithm
undistorted image : test5.jpg
brightness adjusted image : test5.jpg

Also, you can use morphological dilation and erosion to make the edge lines continuous:

```
    kernel = np.ones((5, 5), np.uint8)
    closing = cv2.morphologyEx(binary_mask.astype(np.uint8), cv2.MORPH_CLO
SE, kernel)
```

http://docs.opencv.org/2.4/modules/imgproc/doc/filtering.html?highlight=morphologyex#morphologyex
http://docs.opencv.org/2.4/doc/tutorials/imgproc/opening_closing_hats/opening_closing_hats.html

Here you can find an interesting article about other methods (LDA and LLC) to detect lines in different lighting conditions:
https://otik.uk.zcu.cz/bitstream/handle/11025/11945/Jang.pdf?sequence=1

OpenCV function or other method has been used to correctly rectify each image to a "birds-eye view". Transformed images should be included in the writeup (or saved to a folder) and submitted with the project.

Good job with "birds-eye view"!

Remember that the next step to improve this part of algorithm is to take into account road slope and use dynamic points to transform image to "birds-eye view".

Here are several interesting papers about obtaining and usage of birds-eye view:
http://www.ijser.org/researchpaper%5CA-Simple-Birds-Eye-View-Transformation-Technique.pdf
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3355419/
https://pdfs.semanticscholar.org/4964/9006f2d643c0fb613db4167f9e49462546dc.pdf
https://pdfs.semanticscholar.org/4074/183ce3b303ac4bb879af8d400a71e27e4f0b.pdf

Methods have been used to identify lane line pixels in the rectified binary image. The left and right line have been identified and fit with a curved functional form (e.g., spine or polynomial). Example images with line pixels identified and a fit overplotted should be included in the writeup (or saved to a folder) and submitted with the project.

Well done using histogram and sliding window method to find right and left lane lines!

Here are some good links for other methods described:
https://www.researchgate.net/publication/257291768_A_Much_Advanced_and_Efficient_Lane_Detection_Algorithm_for_Intelligent_Highway_Safety
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5017478/

Algorithm based on current project😄:
https://chatbotslife.com/robust-lane-finding-using-advanced-computer-vision-techniques-46875bb3c8aa#.l2uxq26sn

The following validation criteria can be used to remove incorrect lines:

- lane width is in defined borders
- lane lines have the same concavity
- the second degree parameter of both fits are not too different (ratio of parameters is less than some value)
- ratio of lines curvature is not too big (be careful here with straight lane)
- distance between left and right lines at the base of the image is roughly the same as at the top of the image (in birds-eye view)
- lane curvature, distance from the center, polynomial coefficients and so on.. don't differ a lot from the same values from the previous frame

Here the idea is to take the measurements of where the lane lines are and estimate how much the road is curving and where the vehicle is located with respect to the center of the lane. The radius of curvature may be given in meters assuming the curve of the road follows a circle. For the position of the vehicle, you may assume the camera is mounted at the center of the car and the deviation of the midpoint of the lane from the center of the image is the offset you're looking for. As with the polynomial fitting, convert from pixels to meters.

Radius of curvature looks too big. We expect values around 600-1000m for the curved road.
It seems that the problem is in polynomial coefficients. You have correctly convert `y_eval` from pixels to meters:

```
radius = ((1 + (2 * fit_coeff[0] * y_eval * ym_per_pix + fit_coeff[1]) **
 2) ** 1.5) / np.absolute \
             (2 * fit_coeff[0])
```

But note that polynomial coefficients are still in pixels. You should convert them as well.

Regarding position towards the center you wrote:

> Calculate the center of the lane as the average x value of the 2 lines computed at the step above

But in code you write:

```
return (rightcoord - leftcoord - width) / 2 * xm_per_pix
```

Note that actual lane center you can define if you summarize x-coordinates near the car for the left line and the x-coordinates for the right line and divide it by 2 (actual center of the lane based on determined left and right lines). But you subtract these values from each other `rightcoord - leftcoord`. Check please that everything is good with your offset.

The fit from the rectified image has been warped back onto the original image and plotted to identify the lane boundaries. This should demonstrate that the lane boundaries were correctly identified. An example image with lanes, curvature, and position from center should be included in the writeup (or saved to a folder) and submitted with the project.

Pipeline (video)

The image processing pipeline that was established to find the lane lines in images successfully processes the video. The output here should be a new video where the lanes are identified in every frame, and outputs are generated regarding the radius of curvature of the lane and vehicle position within the lane. The pipeline should correctly map out curved lines and not fail when shadows or pavement color changes are present. The output video should be linked to in the writeup and/or saved and submitted with the project.

Excellent pipeline implementation!

Note please that besides (or inside) `Line` class mentioned in lecture you can try to use exponential smoothing between current and previous values:

```
def smooth(self, prev, curr, coeficient = 0.4):
        '''
```

```
         exponential smoothing
        :param prev: old value
        :param curr: new value
        :param coeficient: smoothing coef.
        :return:
        '''
        return curr*coeficient + prev*(1-coeficient)
```

https://en.wikipedia.org/wiki/Exponential_smoothing

This can be applied to radius of curvature and/or to polynomial coefficients. Just pass to `smooth` method previous (`prev`) and current (`curr`) values you want to smooth. Another way to implement it is to use `Line` class as was mentioned in lectures, store values for curvature and coefficients from the last `n` frames and average them before drawing lane on the image.

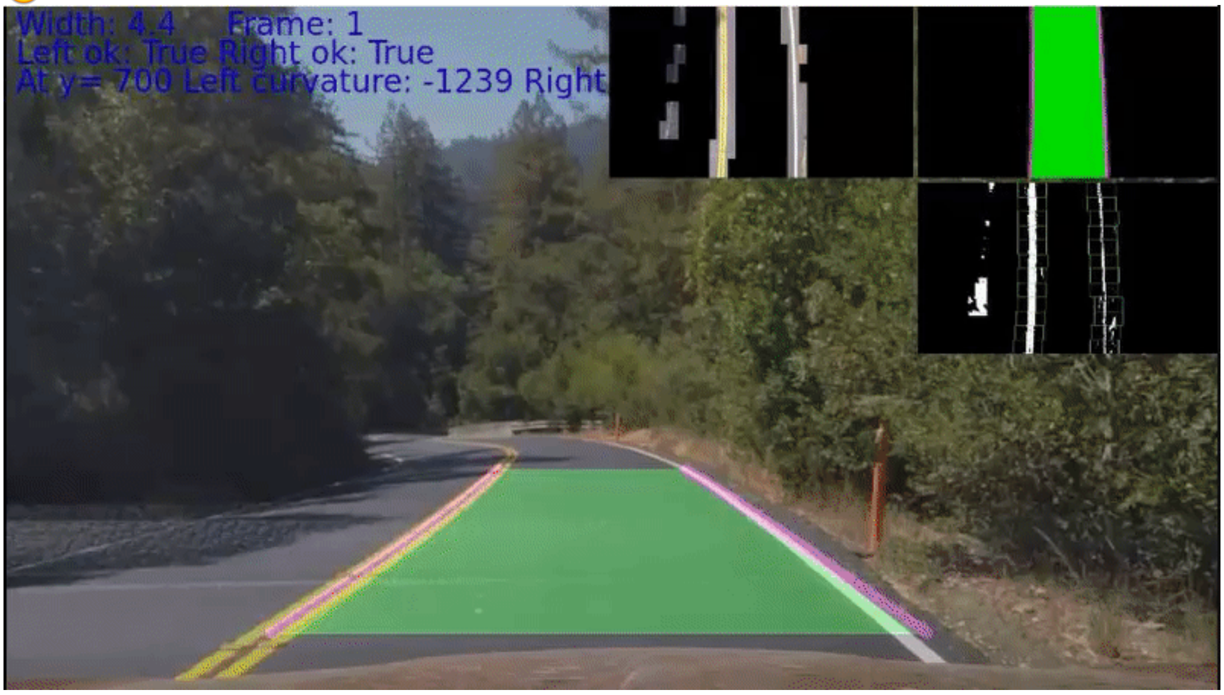
Also for debugging purposes if you add some sanity check/validation criteria mentioned above you can save images where lines are not detected for further analysis and fine tune your algorithm particular for these images. You can use for it the following code for example:

```
if here_is_some_condition_that_indicates_that_sanity_check_fails:
    # found problematic image - cannot find the line at all, save it for l
ater analysis
    fname = str(random.randint(0, 1000000)) + ".jpg"
    outImg = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
    cv2.imwrite(fname, outImg)
    return img
```

If Computer Vision method with its paramters tuning is perfect for current video, it is not good (if not say bad at all) to generalize it for all possible situation on the road. Deep Learning techniques is much better for generalization!

It is difficult to find general parameters for all possible road and weather conditions. One way to resolve this problem is to divide possible conditions into classes and make dynamic threshold parameters for each class. They can depend on weather class, image lighting, shadows, road type and so on. It is a very tough task. For example, I'm not sure that it is possible to find good parameters for harder challenge video. Another way is to build neural network that will learn by itself based on video. This approach works very well for harder challenge video (here is, for example, work one of the nenodegree student - I've increased speed a little bit

😄):


Width: 4.4 Frame: 1
Left ok: True Right ok: True
At y= 700 Left curvature: -1239 Right

Model similar to NVidia from project 3 was applied for DL training.

Remember that in real world you should detect lane in real time. So it can be useful to measure time that take your algorithm (for example how many seconds/milliseconds you spend on one frame) and adjust it to improve these values. But also note please that Python is usually used for prototyping in such class of tasks but in real car where you should process data in real time some precompiled language (C/C++ for example) are used.

Discussion

Discussion includes some consideration of problems/issues faced, what could be improved about their algorithm/pipeline, and what hypothetical cases would cause their pipeline to fail.

Well done with discussion!

RESUBMIT

DOWNLOAD PROJECT

Best practices for your project resubmission

Ben shares 5 helpful tips to get you through revising and resubmitting your project.

Watch Video (3:01)

RETURN TO PATH