

Project01-LaneLines

P16084251 系統所 周禮宏

● Code 的詳細描述解說

引入需要的套件

Import Packages

```
In [1]: #importing some useful packages
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
import cv2
%matplotlib inline
```

各套件的版本

Numpy: 1.16.4

Matplotlib: 3.1.0

Cv2: 3.4.2

Read in an Image

```
In [2]: #reading in an image
image = mpimg.imread('test_images/solidWhiteRight.jpg')

#printing out some stats and plotting
print('This image is:', type(image), 'with dimensions:', image.shape)
plt.imshow(image) # if you wanted to show a single color channel image called 'gray', for example, call a
s plt.imshow(gray, cmap='gray')
```

This image is: <class 'numpy.ndarray'> with dimensions: (540, 960, 3)

Out[2]: <matplotlib.image.AxesImage at 0x1fec314d898>



mpimg.imread() 用來讀取影像

image.shape 可以印出影像的尺寸，此影像為 540 x 960

此外，因為是彩色影像，所以有 3 個 channel (RGB)

plt.show() 用來顯示影像

Helper Functions

Below are some helper functions to help get you started. They should look familiar from the lesson!

```
In [3]: import math

def grayscale(img):
    """Applies the Grayscale transform
    This will return an image with only one color channel
    but NOTE: to see the returned image as grayscale
    (assuming your grayscaled image is called 'gray')
    you should call plt.imshow(gray, cmap='gray')"""
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
    # Or use BGR2GRAY if you read an image with cv2.imread()
    # return cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

def canny(img, low_threshold, high_threshold):
    """Applies the Canny transform"""
    return cv2.Canny(img, low_threshold, high_threshold)

def gaussian_blur(img, kernel_size):
    """Applies a Gaussian Noise kernel"""
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

def region_of_interest(img, vertices):
    """
    Applies an image mask.

    Only keeps the region of the image defined by the polygon
    formed from `vertices`. The rest of the image is set to black.
    `vertices` should be a numpy array of integer points.
    """
    #defining a blank mask to start with
    mask = np.zeros_like(img)

    #defining a 3 channel or 1 channel color to fill the mask with depending on the input image
    if len(img.shape) > 2:
        channel_count = img.shape[2] # i.e. 3 or 4 depending on your image
        ignore_mask_color = (255,) * channel_count
    else:
        ignore_mask_color = 255

    #filling pixels inside the polygon defined by "vertices" with the fill color
    cv2.fillPoly(mask, vertices, ignore_mask_color)

    #returning the image only where mask pixels are nonzero
    masked_image = cv2.bitwise_and(img, mask)
    return masked_image

def draw_lines(img, lines, color=[255, 20, 147], thickness=10):
    """
    NOTE: this is the function you might want to use as a starting point once you want to
    average/extrapolate the line segments you detect to map out the full
    extent of the lane (going from the result shown in raw-lines-example.mp4
    to that shown in P1_example.mp4).

    Think about things like separating line segments by their
    slope ((y2-y1)/(x2-x1)) to decide which segments are part of the left
    line vs. the right line. Then, you can average the position of each of
    the lines and extrapolate to the top and bottom of the lane.

    This function draws `lines` with `color` and `thickness`.
    Lines are drawn on the image inplace (mutates the image).
    If you want to make the lines semi-transparent, think about combining
    this function with the weighted_img() function below
    """
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)
```

```

def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    """
    `img` should be the output of a Canny transform.

    Returns an image with hough lines drawn.
    """
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)
    return line_img

# Python 3 has support for cool math symbols.

def weighted_img(img, initial_img, α=0.8, β=1., γ=0.):
    """
    `img` is the output of the hough_lines(), An image with lines drawn on it.
    Should be a blank image (all black) with lines drawn on it.

    `initial_img` should be the image before any processing.

    The result image is computed as follows:

    initial_img * α + img * β + γ
    NOTE: initial_img and img must be the same shape!
    """
    return cv2.addWeighted(initial_img, α, img, β, γ)

```

幫助影像處理各函式

grayscale(img)

用途：將輸入影像轉換為灰階影像

輸入：img（輸入影像）

輸出：灰階影像

canny(img, low_threshold, high_threshold)

用途：進行邊緣檢測

輸入：

img（輸入影像，必須為單通道的灰階影像）

low_threshold（為較小的閾值，將間斷的邊緣連接起來）

high_threshold（為較大的閾值，檢測圖像中明顯的邊緣）

輸出：邊緣檢測後的影像

gaussian_blur(img, kernel_size)

用途：進行高斯模糊，有降噪的效果

輸入：

img（原始影像）

kernel_size（指定高斯核的寬和高，必須是奇數）

輸出：高斯模糊後的影像

region_of_interest(img, vertices)

用途：選取影像中有興趣的區域，即所在的道路線

輸入：

img (輸入影像)

vertices (選取區域的各頂點)

輸出: 只含有興趣區域的影像

補充說明:

cv2.fillPoly() 可以填充任意形狀的圖型。

```
if len(img.shape) > 2:
    channel_count = img.shape[2]
    ignore_mask_color = (255,) * channel_count
else:
    ignore_mask_color = 255
```

判斷是彩色影像或灰階影像以決定遮罩的顏色。

cv2.bitwise_and() 對影像中每個像素進行二進位「與」的操作，可用來提取精確的邊界。

 draw_lines(img, lines, color=[255,20,147], thickness=10)

用途: 將挑選出邊緣點連成一直線，期望與道路線相符

輸入:

img (輸入影像)

lines (點的座標)

color (線的顏色，預設為粉紅色)

thickness (線的粗度，預設為 10)

輸出: 含線的影像

 hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap)

用途: 檢測影像中的直線或者圓等幾何圖形

輸入:

img (輸入影像)

rho (像素每次疊代的大小)

theta (角度累加器的大小)

threshold (閾值大小，即每個參數對至少經過的像素點數)

min_line_length (輸出中接受的線的最小長度)

max_line_gap (允許連接到線的區段之間的最大距離)

輸出: Hough Transform 後的影像，並繪上線

weighted_img(img, initial_img, $\alpha=0.8$, $\beta=1.$, $\gamma=0.$)

用途：影像疊加

輸入：

img (疊加的影像 1)

initial_img (疊加的影像 2)

α (影像 1 的透明度)

β (影像 2 的透明度)

輸出：兩個影像疊加後的影像

Test Images

Build your pipeline to work on the images in the directory "test_images"

You should make sure your pipeline works well on these images before you try the videos.

```
In [4]: import os
images = os.listdir("test_images/")
```

引入測試的影像集

Build a Lane Finding Pipeline

Build the pipeline and run your solution on all test_images. Make copies into the test_images_output directory, and you can use the images in your writeup report.

Try tuning the various parameters, especially the low and high Canny thresholds as well as the Hough lines parameters.

```
In [5]: # TODO: Build your pipeline that will draw lane lines on the test_images
# then save them to the test_images_output directory.

def process_image(image):
    # NOTE: The output you return should be a color image (3 channel) for processing video below
    # TODO: put your pipeline here,
    # you should return the final output (image where lines are drawn on lanes)


    gray_image = grayscale(image)
    gaus_blur = gaussian_blur(gray_image, 5)
    edges = canny(gaus_blur, 50, 150)
    imshape = image.shape

    vertices = np.array([(0, imshape[0]), (480, 320), (500, 320), (imshape[1], imshape[0])], dtype=np.int32)
    masked = region_of_interest(edges, vertices)

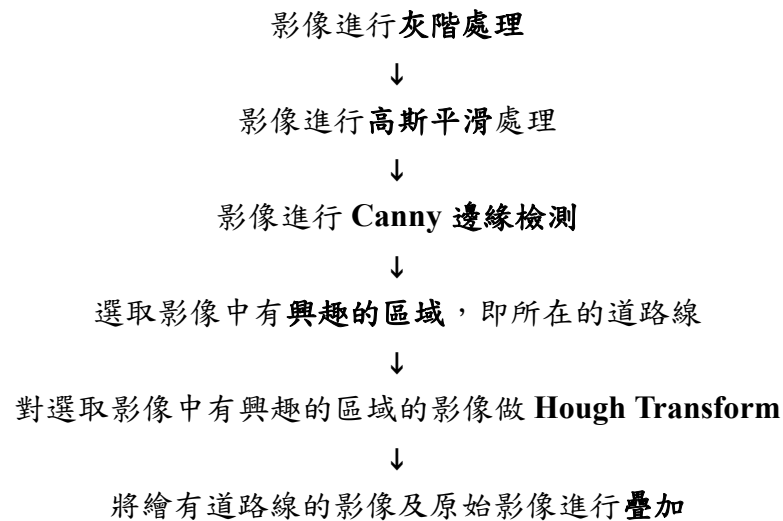
    rho = 2 #distance resolution in pixels of the Hough grid
    theta = np.pi/180 #angular resolution in radians of the Hough grid
    threshold = 5 #minimum number of votes (intersections in Hough grid cell)
    min_line_len = 10 #minimum number of pixels making up a line
    max_line_gap = 20 #maximum gap in pixels between connectable line segments
    line_image = hough_lines(masked, rho, theta, threshold, min_line_len, max_line_gap)

    result = weighted_img(line_image, image)


    return result
```

 process_image(image)

流程：



重要參數設定：

 gaussian_blur(img, kernel_size)

kernel_size = 5

 canny(img, low_threshold, high_threshold)

low_threshold = 50

high_threshold = 150

 region_of_interest(img, vertices)

vertices = np.array([[(0,imshape[0]),(450, 320), (500, 320),
(imshape[1],imshape[0])]], dtype=np.int32)

 hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap)

rho = 2

theta = np.pi/180

threshold = 5

min_line_len = 10

max_line_gap = 20

```

for img_file in images:
    #print(img_file)
    # Skip all files starting with line.
    if img_file[0:4] == 'line':
        continue

    image = mpimg.imread('test_images/' + img_file)

    weighted = process_image(image)

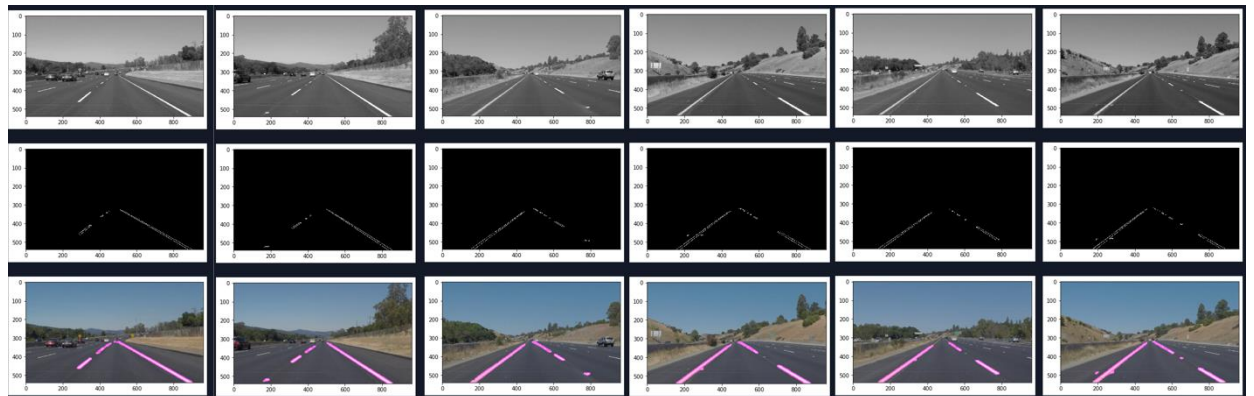
    plt.imshow(weighted)
    #break
    mpimg.imsave('test_images/lines-' + img_file, weighted)

```



對影像集進行影像處理，並儲存輸出影像

各影像處理階段比較（上圖:灰階影像、中圖:道路線提取、下圖:輸出影像）



Test on Videos

You know what's cooler than drawing lanes over images? Drawing lanes over video!

We can test our solution on two provided videos:

solidWhiteRight.mp4

solidYellowLeft.mp4

Note: if you get an import error when you run the next cell, try changing your kernel (select the Kernel menu above --> Change Kernel). Still have problems? Try relaunching Jupyter Notebook from the terminal prompt. Also, consult the forums for more troubleshooting tips.

If you get an error that looks like this:

```
NeedDownloadError: Need ffmpeg exe.  
You can download it by calling:  
imageio.plugins.ffmpeg.download()
```

Follow the instructions in the error message and check out [this forum post](#) for more troubleshooting tips across operating systems.

```
In [6]: # Import everything needed to edit/save/watch video clips  
from moviepy.editor import VideoFileClip  
from IPython.display import HTML
```

對影片進行影像處理

引入處理影片需要的套件

各套件版本:

moviepy: 1.0.1

ipython: 7.6.1

Let's try the one with the solid white lane on the right first ...

```
In [8]: white_output = 'test_videos_output/solidWhiteRight.mp4'  
## To speed up the testing process you may want to try your pipeline on a shorter subclip of the video  
## To do so add .subclip(start_second,end_second) to the end of the line below  
## Where start_second and end_second are integer values representing the start and end of the subclip  
## You may also uncomment the following line for a subclip of the first 5 seconds  
##clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4").subclip(0,5)  
clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4")  
white_clip = clip1.fl_image(process_image) #NOTE: this function expects color images!!  
%time white_clip.write_videofile(white_output, audio=False)
```

```
Moviepy - Building video test_videos_output/solidWhiteRight.mp4.  
Moviepy - Writing video test_videos_output/solidWhiteRight.mp4
```

```
Moviepy - Done !  
Moviepy - video ready test_videos_output/solidWhiteRight.mp4  
Wall time: 3.48 s
```

Play the video inline, or if you prefer find the video in your filesystem (should be in the same directory) and play it in your video player of choice.

```
In [9]: HTML("""  
<video width="960" height="540" controls>  
  <source src="{0}">  
</video>  
""").format(white_output)
```

```
Out[9]: <IPython.core.display.HTML object>
```

對 solidWhiteRight.mp4 影片進行影像處理

VideoFileClip() 將影片匯入

clip1.fl_image(process_image) 進行影像處理

white_clip.write_videofile(white_output, audio=False) 儲存輸出影片


```
In [10]: yellow_output = 'test_videos_output/solidYellowLeft.mp4'
        ## To speed up the testing process you may want to try your pipeline on a shorter subclip of the video
        ## To do so add .subclip(start_second,end_second) to the end of the line below
        ## Where start_second and end_second are integer values representing the start and end of the subclip
        ## You may also uncomment the following line for a subclip of the first 5 seconds
        ##clip2 = VideoFileClip('test_videos/solidYellowLeft.mp4').subclip(0,5)
        clip2 = VideoFileClip('test_videos/solidYellowLeft.mp4')
        yellow_clip = clip2.fl_image(process_image)
        %time yellow_clip.write_videofile(yellow_output, audio=False)

Moviepy - Building video test_videos_output/solidYellowLeft.mp4.
Moviepy - Writing video test_videos_output/solidYellowLeft.mp4

Moviepy - Done !
Moviepy - video ready test_videos_output/solidYellowLeft.mp4
Wall time: 9.41 s
```

```
In [12]: HTML("""
        <video width="960" height="540" controls>
        <source src="{0}">
        </video>
        """).format(yellow_output)
```

Out[12]:

對 solidYellowLeft.mp4 影片進行影像處理
步驟如上

Optional Challenge

Try your lane finding pipeline on the video below. Does it still work? Can you figure out a way to make it more robust? If you're up for the challenge, modify your pipeline so it works with this video and submit it along with the rest of your project!

```
In [13]: def process_image_challenge(image):
        # NOTE: The output you return should be a color image (3 channel) for processing video below
        # TODO: put your pipeline here,
        # you should return the final output (image where lines are drawn on lanes)

        gray_image = grayscale(image)
        gaus_blur = gaussian_blur(gray_image, 5)
        edges = canny(gaus_blur, 50,250)
        imshape = image.shape

        vertices = np.array([(270,650),(580, 460), (730, 460), (1150,650)]), dtype=np.int32)
        masked = region_of_interest(edges, vertices)

        rho = 0.5 #distance resolution in pixels of the Hough grid
        theta = np.pi/180 #angular resolution in radians of the Hough grid
        threshold = 10 #minimum number of votes (intersections in Hough grid cell)
        min_line_len = 90 #minimum number of pixels making up a line
        max_line_gap = 200 #maximum gap in pixels between connectable line segments

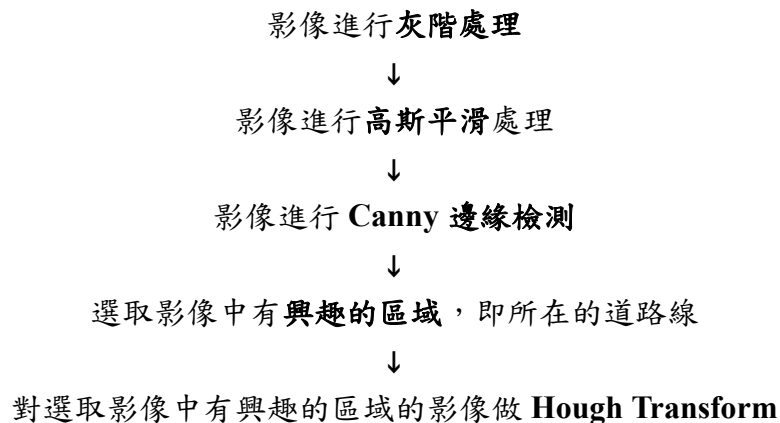
        line_image = hough_lines(masked, rho, theta, threshold, min_line_len, max_line_gap)

        result = weighted_img(line_image, image)

        return result
```

 process_image_challenge(image)

流程：





將繪有道路線的影像及原始影像進行疊加

重要參數設定：

由於 challenge.mp4 影片的車道範圍與上面影片大不相同，加上有陰影及彎道的變化，所以參數需另外設定。

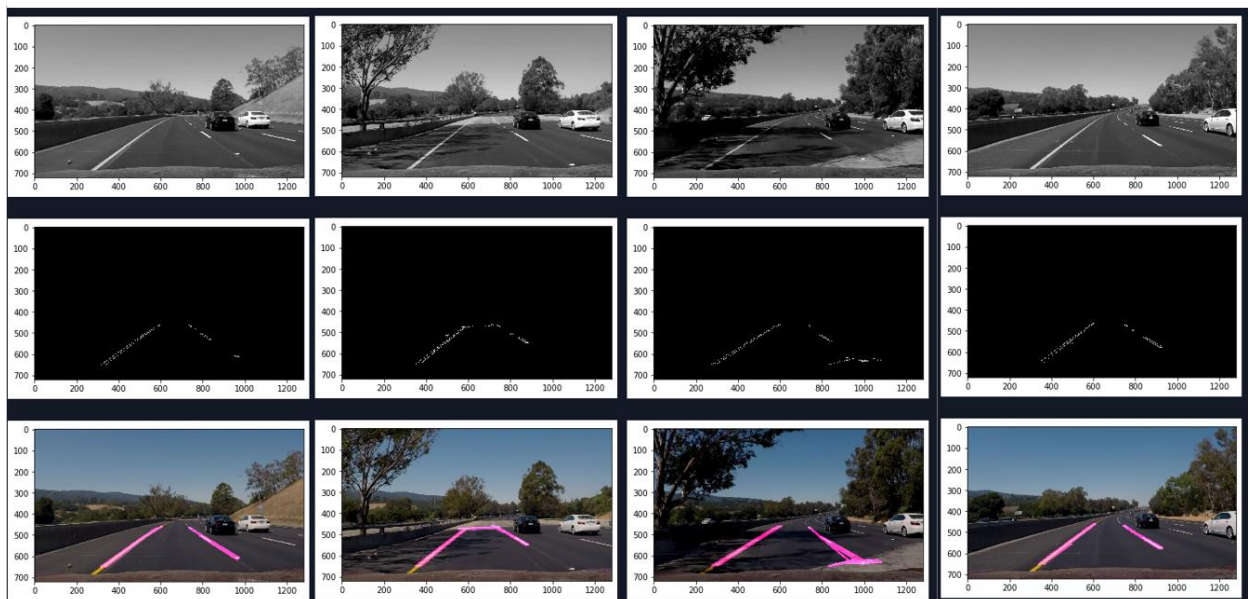
```
gaussian_blur(img, kernel_size)
kernel_size = 5
```

```
canny(img, low_threshold, high_threshold)
low_threshold = 50
high_threshold = 250
```

```
region_of_interest(img, vertices)
vertices = np.array([(270,650),(580, 460), (730, 460), (1150,650)]), dtype=np.int32)
```

```
hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap)
rho = 0.5
theta = np.pi/180
threshold = 10
min_line_len = 90
max_line_gap = 200
```

各影像處理階段比較（上圖：灰階影像、中圖：道路線提取、下圖：輸出影像）



```

In [14]: challenge_output = 'test_videos_output/challenge.mp4'
         ## To speed up the testing process you may want to try your pipeline on a shorter subclip of the video
         ## To do so add .subclip(start_second,end_second) to the end of the line below
         ## Where start_second and end_second are integer values representing the start and end of the subclip
         ## You may also uncomment the following line for a subclip of the first 5 seconds
         ##clip3 = VideoFileClip('test_videos/challenge.mp4').subclip(0,5)
         clip3 = VideoFileClip('test_videos/challenge.mp4')
         challenge_clip = clip3.fl_image(process_image_challenge)
         %time challenge_clip.write_videofile(challenge_output, audio=False)

Moviepy - Building video test_videos_output/challenge.mp4.
Moviepy - Writing video test_videos_output/challenge.mp4

Moviepy - Done !
Moviepy - video ready test_videos_output/challenge.mp4
Wall time: 6.96 s

In [15]: HTML("""
         <video width="960" height="540" controls>
         <source src="{0}">
         </video>
         """,format(challenge_output))

Out[15]:

```

對 challenge.mp4 影片進行影像處理並儲存輸出影片

Github 完整程式碼：

https://github.com/howard1352h/Self-driving/blob/master/P1_V1.ipynb

● 輸出影片 (Youtube 連結)

solidWhiteRight 輸出影片：

<https://www.youtube.com/watch?v=OjsSSQIJ3Wg>

solidYellowLeft 輸出影片：

https://www.youtube.com/watch?v=-g_zKQO8okQ

challenge 輸出影片：

<https://www.youtube.com/watch?v=r20tMvcOiSM>

● 可以改善之處

從影片可以看出此處理方法對於光線的強度變化相當敏感，例如經過陰影之處時，會將陰影邊界誤判成道路線，希望日後再加入其他處理來解決這個問題。