國立成功大學
自動駕駛實務 Project 1 報告

Finding Lane Lines of the Road

學生: 陳育政

學號: E24094198

指導教授: 傅志雄、莊智清

大綱

一、前言

二、實作方法

1. Images Extraction
2. Load Images and Show Images
3. Color Selection
4. Canny Edge Detection
5. Region of Interest Selection (ROI)
6. Hough Transform Line Detection
7. Averaging and Extrapolating Lane Lines
8. Apply Pipeline to Video Clips

三、問題討論

四、結論

五、參考資料

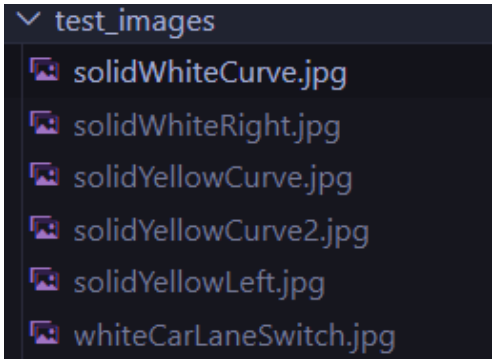
一、 前言

這次的 Project 1 主要是要利用上課教到的一些影像處理技巧組成一個 Pipeline，針對一段影片的每一幀畫面進行處理，最後將所有畫面組回去一個完整的影片，進而實現利用 Python 及 OpenCV 辨識車道的功能。而辨識車道的 Pipeline 主要由 Color Selection、Canny Edge Detection、Region of Interest Selection (ROI)以及 Hough Transform Line Detection 達成。最後，需要額外考慮將偵測到的 Line 進行平均及延伸的動作，車道在影片中的呈現才會相對穩定正常。

二、實作方法

1. Images Extraction

- 在用 Pipeline 處理影片之前，我們要先確定怎樣的 Pipeline 組合會有最佳的效果，因此為了觀察方便，我們要先擷取幾張影像，嘗試不同組合的結果。
- Udacity 這個 GitHub 專案已經有 6 張 images 存在 test_images/ 這個資料夾底下，如右圖所示:



- 但是針對 bonus.mp4 的影片，我們沒有現成的 images，所以我利用 moviepy 的 function 擷取 6 張影像，分別為 4, 8, 12, 16, 20, 24 秒，並另外存在 bonus_images/ 資料夾中，擷取 images 的程式碼如下圖所示:

```
import os
from moviepy.editor import *

# this function will extract frames from the bonus.mp4 and save the jpg to bonus_images directory
def extract_frames(movie, times, imgdir):
    clip=VideoFileClip(movie)
    for t in times:
        imgpath=os.path.join(imgdir, '{}.jpg'.format(t))
        clip.save_frame(imgpath, t)

# extract the frames by calling the extract_frames function
movie='test_videos/bonus.mp4'
imgdir='bonus_images'
times=4,8,12,16,20,24

extract_frames(movie, times, imgdir)
```

2. Load Images and Show Images

- 因為我們目前要先針對 images 做影像處理，所以先將共 12 張 images 載入，並顯示出來，確認 images 有被正確讀入。
- test_images/*.jpg

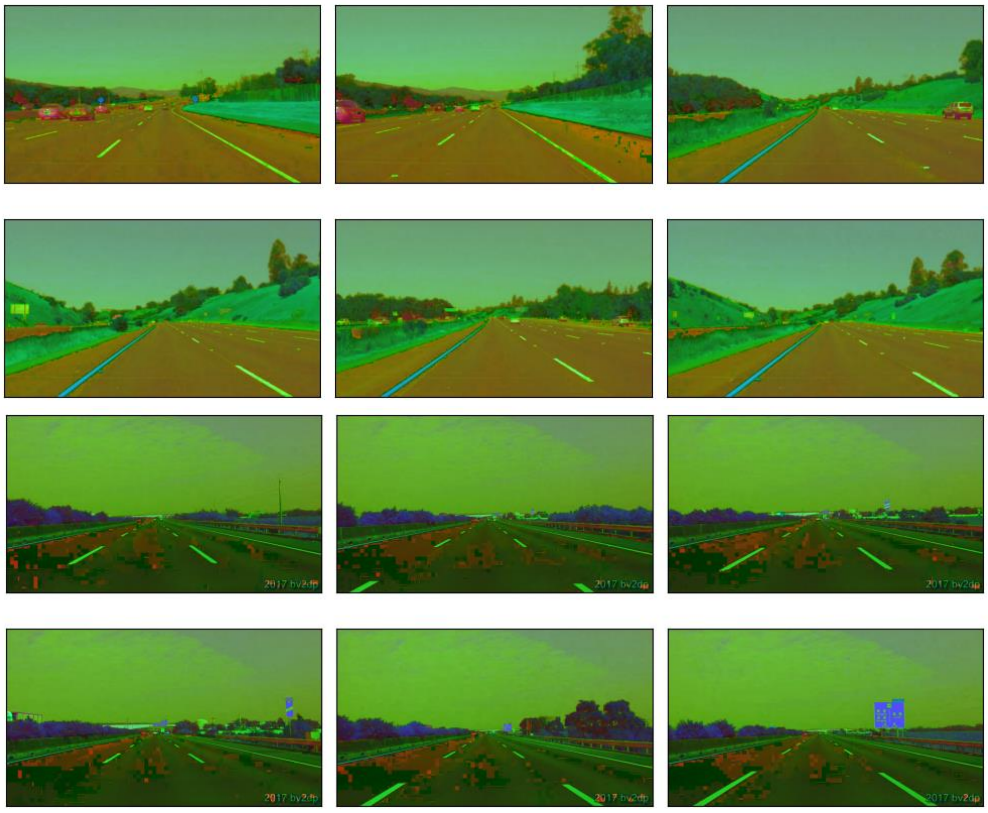
- bonus_images/*.jpg

- 我們可以從 12 張 images 觀察到車道的顏色都是白色或是黃色，以及都是位於 image 的中間下方，還有有些車道線是非連續的，這些因素都要在之後的 pipeline 中考慮到。

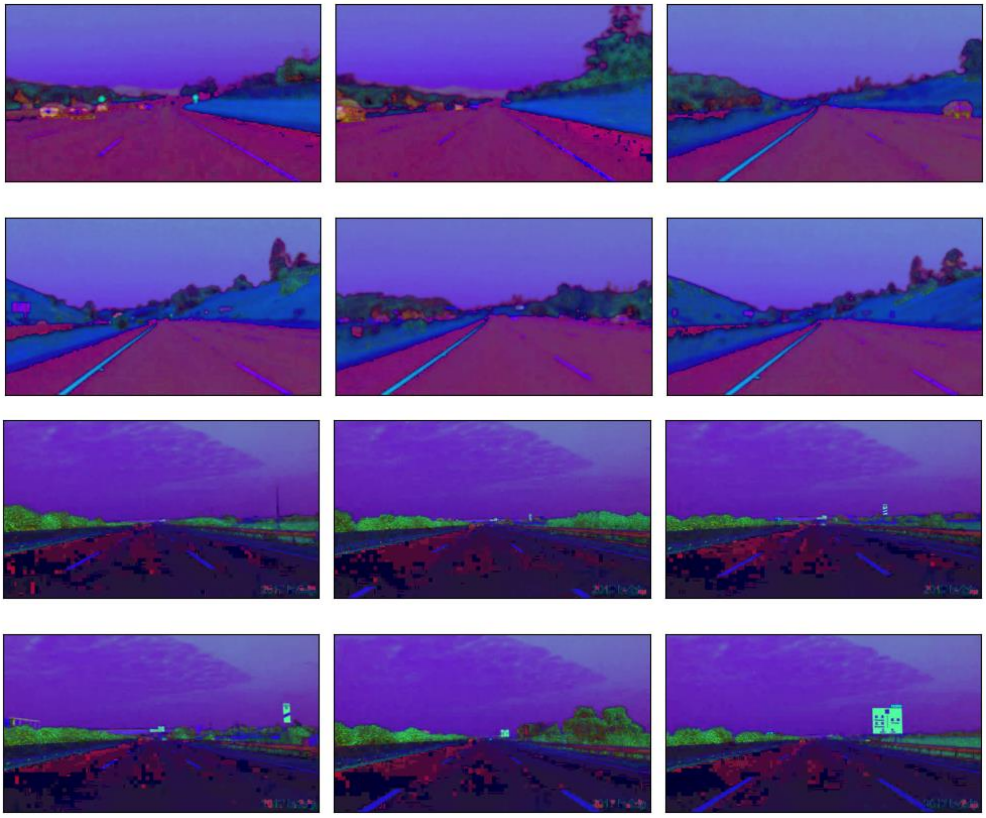
3. Color Selection

- 目前常見的 Color Space 有 RGB、HSL 以及 HSV 共 3 種。RGB 就是由 Red/Green/Blue 三個頻段決定顏色，就是我們一般看到的彩色畫面，HSL 則是由 Hue/Saturation/Lightness 決定顏色，而 HSV 則是 Hue/Saturation/Value。
- 以下將會把 images 的 color space 做轉換，觀察最能夠凸顯黃色及白色的 color space，方便進行之後的 color selection。

➢ 下圖是針對 images 做 HSL 轉換的結果:



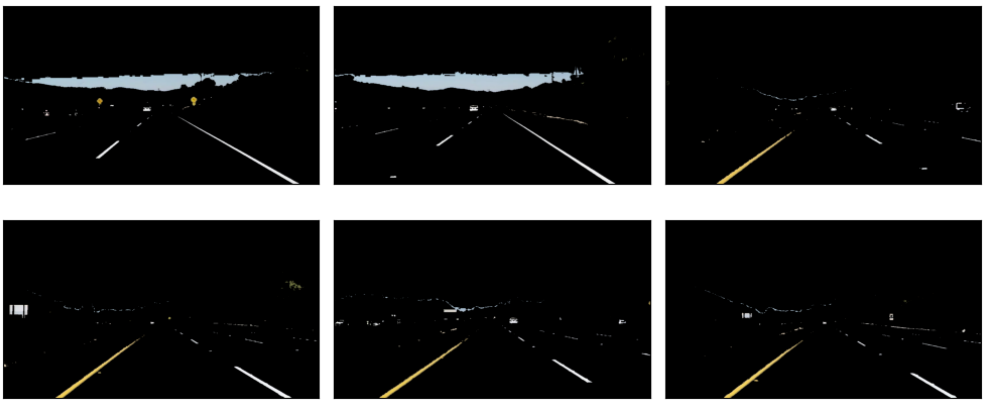
➢ 下圖是針對 images 做 HSV 轉換的結果



- 我們可以透過觀察上圖發現當我們對於 images 做 HSV 的 color space 轉換後，白色車道會變得不明顯，因此可以推斷之後的 color selection 效果會不如 HSL 的好。
- 以下為了版面精簡，因此將依序只放上 test_images/*.jpg 轉換成 RGB/HSL/HSV color space 後 color selection 的結果。
- RGB color selection result:

- HSL color selection result:

- HSV color selection result:

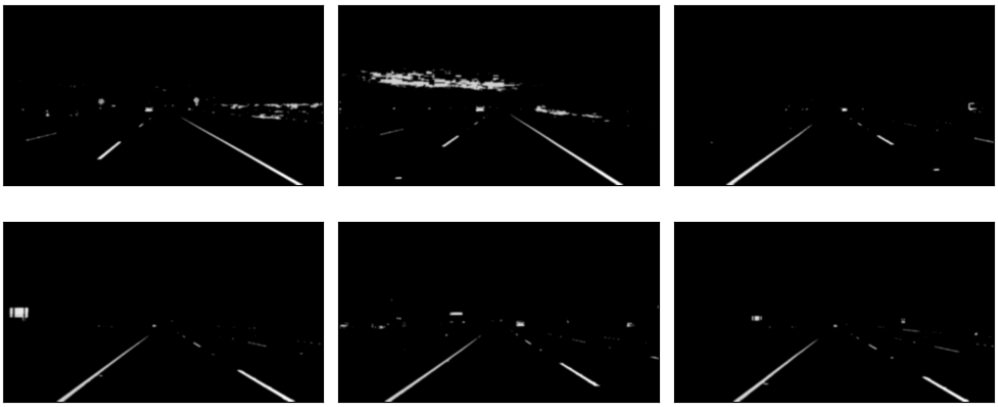


- 觀察上面 3 組不同 color space 下的 color selection 結果，可以發現 HSL 的結果比其他兩者好。因為 HSL 的結果黃色和白色能被凸顯出來，因此之後的 pipeline 使用 HSL 作為 color selection 的依據。

4. Canny Edge Detection

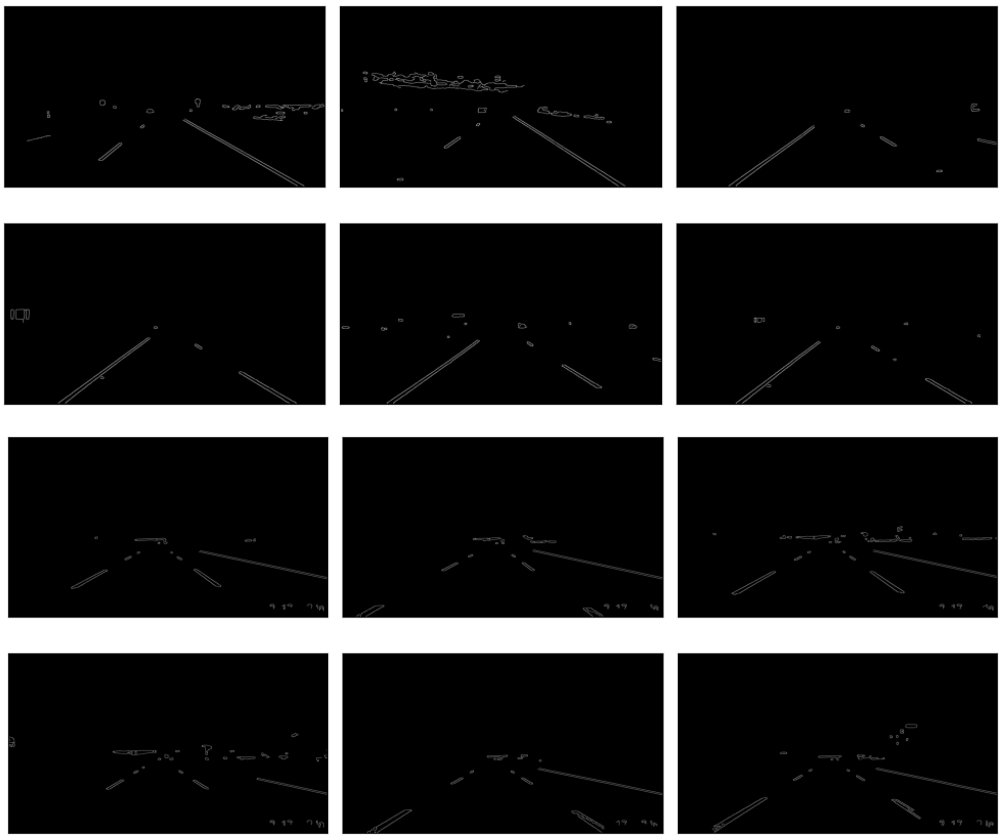
- Canny Edge Detection 是藉由計算灰階圖片的 pixel 梯度變化，進而找出圖片中物體輪廓或是邊界，為了避免邊界過於不平整，因此我們會先使用 Gaussian Blur，使 edge 更平滑。
- 以下依序列出當 test_images/*.jpg 施加 Gaussian Blur 的 kernel size 為 7, 11, 15 時的結果。因為當 kernel size 越大時，所需要花費的處理時間越長，因此我們應該挑選相對較小且能有效模糊圖片的 kernel size。最後，我選用 kernel size=11 進行 Gaussian Blur。
- kernel size=7

- kernel size=11



- kernel size=15

- 接著，我把 Gaussian Blur 後的 image 進行 Canny Edge Detection，並且根據 OpenCV 官方文件所述，Canny 本人推薦 high threshold 和 low threshold 的比例應該為 3 倍，因此我將 high threshold 設為 150，low threshold 則設為 50。
- Edge Detection 結果如下所示:

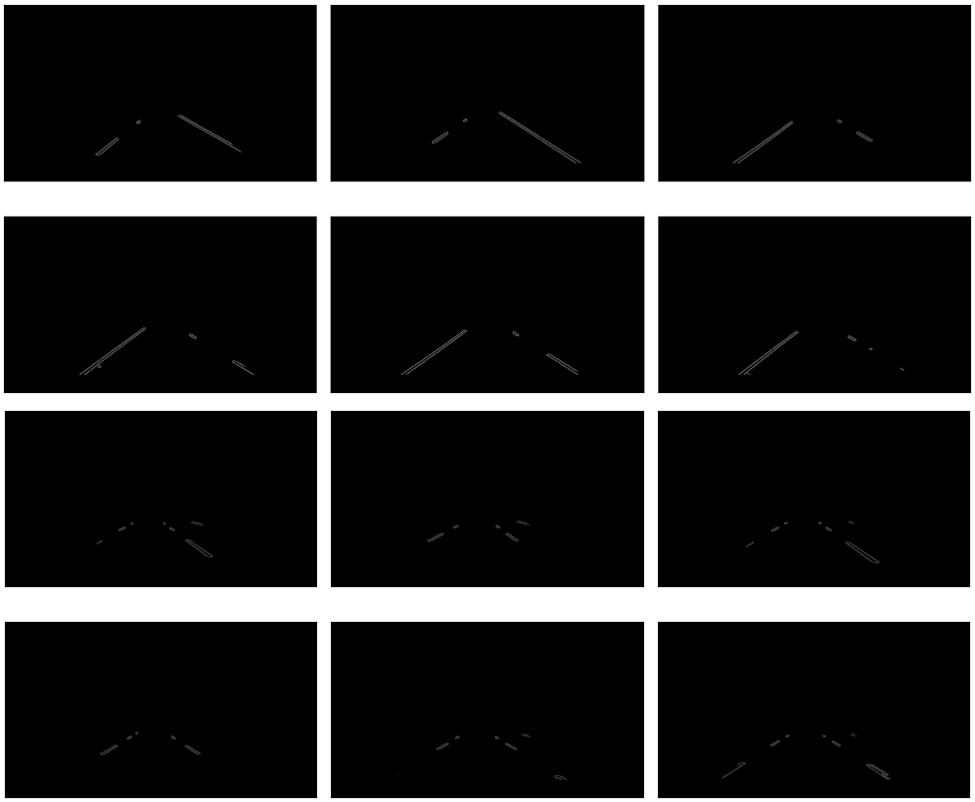


5. Region of Interest Selection (ROI)

- 觀察上面 Edge Detection 之後的結果，我們發現實際上我們需要的車道只有中間下方的區塊，其餘的邊界線條並不是我們所在意的重點。因此，利用 cv2 的 fillPoly 函式隱藏那些不重要的 edges，只留下一個位於圖片中間下方的梯形。
- 因為不同 image 的 size 不同，所以為了讓程式碼更加 general，所以 row 及 col 的數值是使用比例的關係決定要留下邊界的區塊，而不是以純量的數字表示。
- 程式碼如下所示:

```
def select_region(image):
    rows, cols=image.shape[:2]
    # define the vertices of the polygon
    bottom_left=[cols*0.2, rows*0.9]
    top_left=[cols*0.4, rows*0.6]
    bottom_right=[cols*0.8, rows*0.9]
    top_right=[cols*0.6, rows*0.6]
    vertices=np.array([[bottom_left, top_left, top_right, bottom_right]], dtype=np.int32)
    return filter_region(image, vertices)
```

- 經過 ROI 處理後的圖片(只剩車道的輪廓)如下:

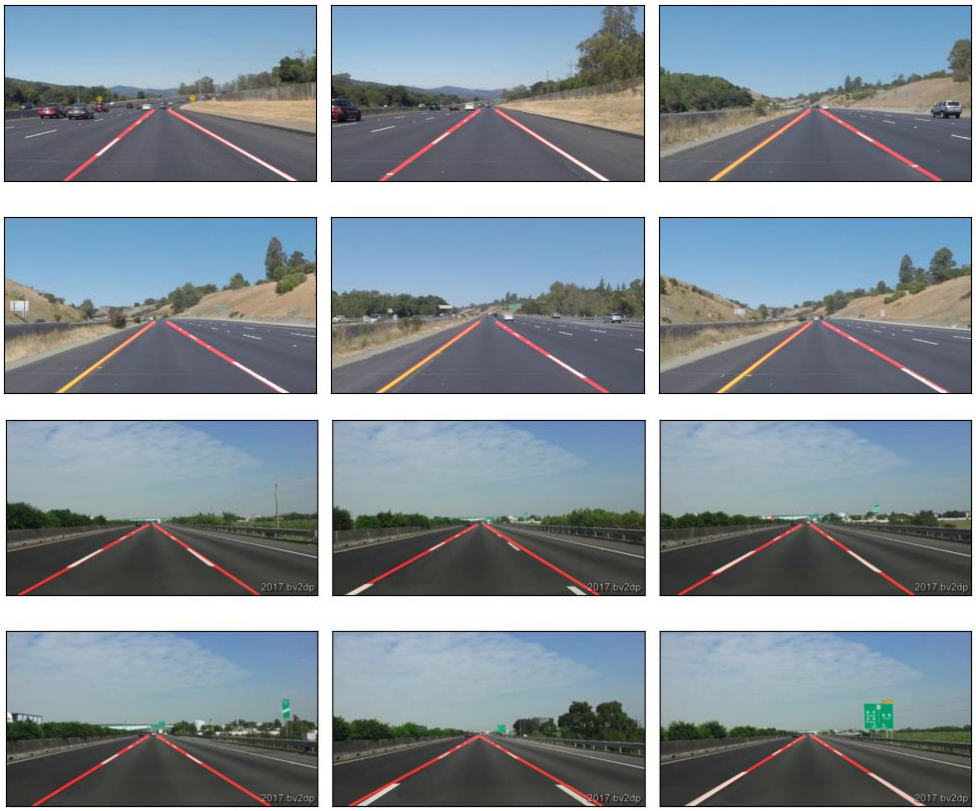


6. Hough Transform Line Detection

- 利用 Hough Transform 將上述 ROI 結果中的邊界，畫成線條，並把線條加在原始圖片上，方便觀察是否車道線有被正確判讀。

7. Averaging and Extrapolating Lane Lines

- 從上一步驟的結果可以看到，我們標示出來的車道線其實有很多條，原因是一條車道線的左右兩側都會形成一條線，因此我在這步驟要做的就是把 averaging lane lines。此外，有些線條是不連續的，所以在 averaging lane lines 之後，要再把線條延伸，也就是 extrapolating lane lines。

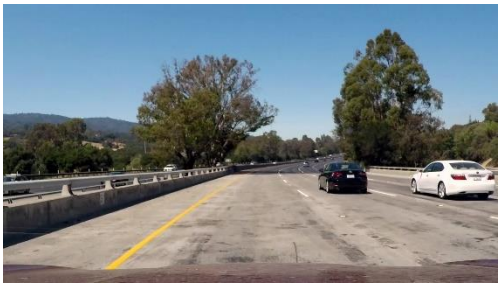


8. Apply Pipeline to Video Clips

- 經過前面的 pipeline 處理，現在可以成功地把圖片中的車道線標示出來，接著就是針對影片的每幀畫面用 pipeline 進行處理，最後再組回成一個加入車道辨識的影片。
- 目前出來的影片結果 solidWhiteRight.mp4 和 solidYellowLeft.mp4 都可以正確辨識車道線，而 challenge.mp4 和 bonus.mp4 有些情況下的車道線會有誤判的情況，下一節的結果與討論將會針對問題進行分析及嘗試解決誤判情況。

三、問題討論

- 當我執行第一版的程式碼後，發現大多數車道線都能被成功辨識並標記，但是有幾種情況下的車道線會誤判，造成飄移不精準的非預期效果。以下將針對幾種情況進行簡單介紹:
- **地板顏色偏淺色及有陰影遮擋**




- **解決方法:**

我擷取三張有上述狀況的影像，並且重新用 pipeline 對影像處理顯示車道線，可以由下圖觀察到其實車道線並沒有因為陰影或是路面淺色而導致嚴重的誤判，反而都有依序畫在車道線上。



於是，我又想到另一種可能就是影片處理的 QUEUE_LENGTH 設定太大了，導致只要當中有幾張幀數的判斷有偏移，則會影響的幀數會偏多，而且校正回正確車道線上所需的時間也隨之大幅提升。因此我便把 QUEUE_LENGTH 由原先設定的 50 改為 10 幀，而最後的影片結果也的確大幅降低原先陰影或路面淺色所造成的誤判。

- **影片右下角有白色字體及最外側車道線的干擾，如下圖所示:**

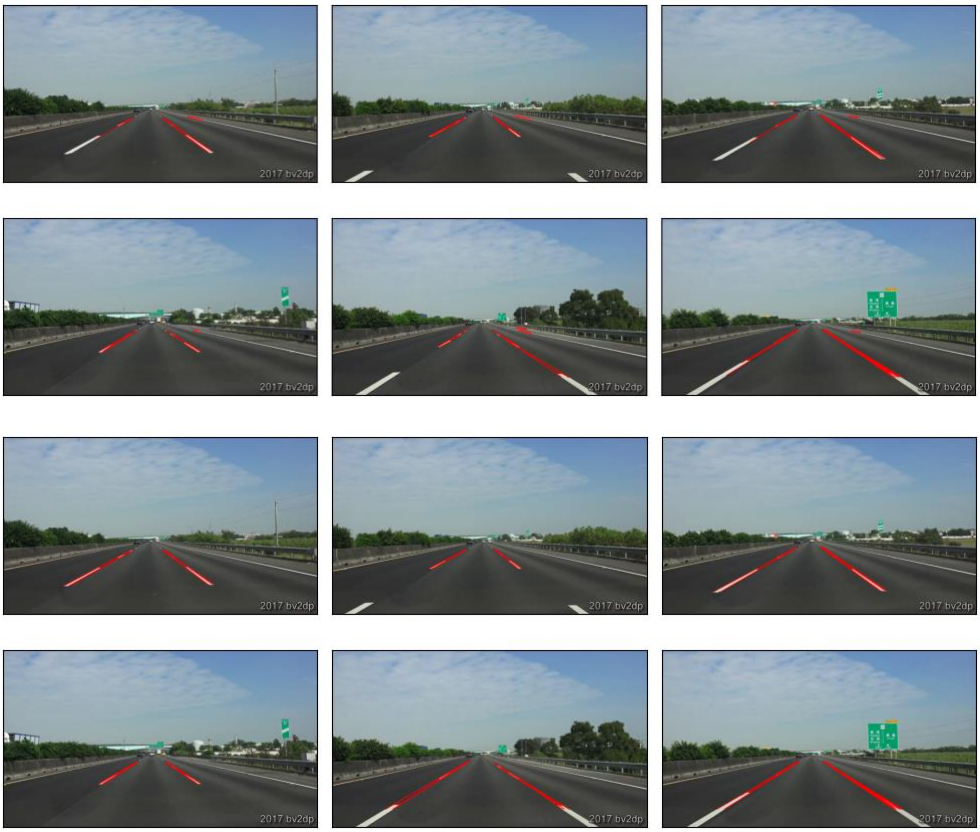
➢ **嘗試解決方法:**

因為我懷疑是影片右下角的白色時間戳記和右上角的最外側車道線干擾了目標車道線的判斷，因此我嘗試了不同的 ROI 參數設定，如下圖所示:

```
def select_region_original(image):
    rows, cols=image.shape[:2]
    # define the vertices of the polygon
    bottom_left=[cols*0.2, rows*0.9]
    top_left=[cols*0.4, rows*0.6]
    bottom_right=[cols*0.8, rows*0.9]
    top_right=[cols*0.6, rows*0.6]
    vertices=np.array([[bottom_left, top_left, top_right, bottom_right]], dtype=np.int32)
    return filter_region_and_show_polygon(image, vertices)


def select_region_after(image):
    rows, cols=image.shape[:2]
    # define the vertices of the polygon
    bottom_left=[cols*0.1, rows*0.9]
    top_left=[cols*0.4, rows*0.6]
    bottom_right=[cols*0.9, rows*0.9]
    top_right=[cols*0.5, rows*0.6]
    vertices=np.array([[bottom_left, top_left, top_right, bottom_right]], dtype=np.int32)
    return filter_region_and_show_polygon(image, vertices)
```

而最後發現是最外側車道的影響比較大，原先的 ROI 設定已經能夠有效的把時間戳記覆蓋掉，以下將附上 select_region_original(上圖)和 select_region_after(下圖)所標示出的車道線結果圖。



我們可以清楚觀察到 select_region_original 的結果會包含一小段的最外側車道線，而正是因為這個原因，導致最後平均及延伸後的結果會有偏移，而藉由改變 ROI 參數，便能有效解決此問題。

四、結論

- **未來可優化部分:**

  我覺得目前的整個 pipeline 來說，Region of Interest(ROI)是比較容易有問題的，舉例來說，原先的 ROI 參數設定對於 Udacity 附上的影片檔，可以完美辨識車道線。但是，一旦套用到 bonus.mp4，就會因為有其他外側車道線的干擾，而導致車道線辨識有誤。

  對於這種問題，當然可以像我這次手動調 ROI 參數讓多餘的車道線消失，但是並不是一種長遠之計。因為有可能車道線若是較靠近物外側，就會被 mask 遮住，換言之，就是手動調參數的做法廣泛性不足。

  所以，我認為或許有另一種想法就是先用 ROI 遮住大部分多餘的邊界(e.g. 天空、兩側風景...等)，接著，在針對處理後的影片分析，只留下最靠近影片視線正中間的兩條斜率相反的直線，這兩條直線便能被視為左右的車道線，同時也忽略掉較外側其他車道線的干擾。

- **成果影片網址:**
  1. solidWhiteRight.mp4: https://youtu.be/mtegFnVVhmA
  2. solidYellowLeft.mp4: https://youtu.be/YPoSxomYsCs
  3. challenge.mp4: https://youtu.be/cnWfqpKX6Go
  4. bonus.mp4: https://youtu.be/xT6pf7xkUus
- **程式碼專案網址:** https://github.com/project1-source-code

五、參考資料

1. RGB/HSL/HSV Introduction
2. Canny Edge Detection Official Docs
3. https://naokishibuya.medium.com/finding-lane-lines-on-the-road
4. https://github.com/santhoshpkumar/FindingLaneLinesOnTheRoad/