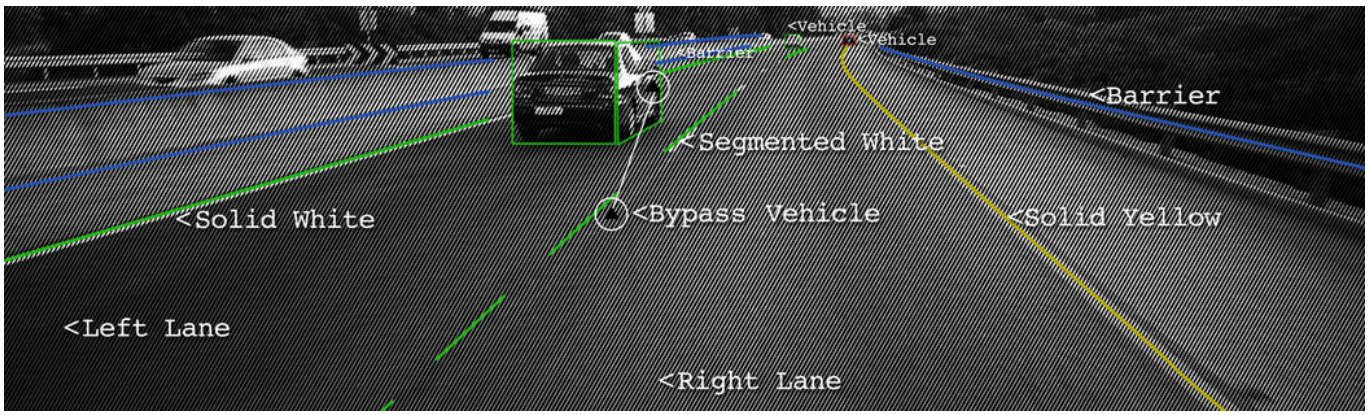
Tutorial: Build a lane detector

Chuan-en Lin 林傳恩 Follow
Dec 17, 2018 · 10 min read ★

Waymo's self-driving taxi service just hit the road this month — but how do autonomous vehicles even work? The lines drawn on roads indicate to human drivers where the lanes are and act as a guiding reference to which direction to steer the vehicle accordingly and convention to how vehicle agents interact harmoniously on the road. Likewise, the ability to identify and track lanes is cardinal for developing algorithms for driverless vehicles.

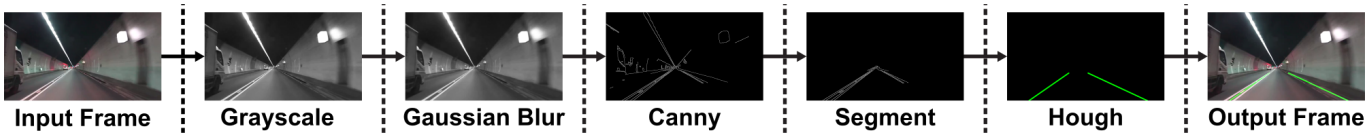


In this tutorial, we will learn how to build a software pipeline for tracking road lanes using computer vision techniques. We will approach this task through two different approaches.

Table of Contents:

Approach 1: Hough Transform

Most lanes are designed to be relatively straightforward not only as to encourage orderliness but also to make it easier for human drivers to steer vehicles with consistent speed. Therefore, our intuitive approach may be to first detect prominent straight lines in the camera feed through edge detection and feature extraction techniques. We will be using OpenCV, an open source library of computer vision algorithms, for implementation. The following diagram is an overview of our pipeline.



Before we start, here is a demo of our outcome:

1. Setting up your environment

If you do not already have OpenCV installed, open Terminal and run:

```
pip install opencv-python
```

Now, clone the tutorial repository by running:

```
git clone https://github.com/chuanenlin/lane-detector.git
```

Next, open `detector.py` with your text editor. We will be writing all of the code of this section in this Python file.

2. Processing a video

We will feed in our sample video for lane detection as a series of continuous frames (images) by intervals of 10 milliseconds. We can also quit the program anytime by pressing the 'q' key.

```
import cv2 as cv

# The video feed is read in as a VideoCapture object
cap = cv.VideoCapture("input.mp4")
while (cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being proje
    ret, frame = cap.read()
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loop
    if cv.waitKey(10) & 0xFF == ord('q'):
        break

# The following frees up resources and closes all windows
cap.release()
cv.destroyAllWindows()
```

checkpoint1.py hosted with ♡ by GitHub view raw

3. Applying Canny Detector

The Canny Detector is a multi-stage algorithm optimized for fast real-time edge detection. The fundamental goal of the algorithm is to detect sharp changes in luminosity (large gradients), such as a shift from white to black, and defines them as edges, given a set of thresholds. The Canny algorithm has four main stages:

A. Noise reduction

As with all edge detection algorithms, noise is a crucial issue that often leads to false detection. A 5x5 Gaussian filter is applied to convolve (smooth) the image to lower the

detector's sensitivity to noise. This is done by using a kernel (in this case, a 5x5 kernel) of normally distributed numbers to run across the entire image, setting each pixel value equal to the weighted average of its neighboring pixels.

5x5 Gaussian kernel. Asterisk denotes convolution operation.

B. Intensity gradient

The smoothened image is then applied with a Sobel, Roberts, or Prewitt kernel (Sobel is used in OpenCV) along the x-axis and y-axis to detect whether the edges are horizontal, vertical, or diagonal.

Sobel kernel for calculation of the first derivative of horizontal and vertical directions

C. Non-maximum suppression

Non-maximum suppression is applied to "thin" and effectively sharpen the edges. For each pixel, the value is checked if it is a local maximum in the direction of the gradient calculated previously.

Non-maximum suppression on three points

A is on the edge with a vertical direction. As gradient is normal to the edge direction, pixel values of B and C are compared with pixel values of A to determine if A is a local maximum. If A is local maximum, non-maximum suppression is tested for the next point. Otherwise, the pixel value of A is set to zero and A is suppressed.

D. Hysteresis thresholding

After non-maximum suppression, strong pixels are confirmed to be in the final map of edges. However, weak pixels should be further analyzed to determine whether it constitutes as edge or noise. Applying two pre-defined minVal and maxVal threshold values, we set that any pixel with intensity gradient higher than maxVal are edges and any pixel with intensity gradient lower than minVal are not edges and discarded. Pixels with intensity gradient in between minVal and maxVal are only considered edges if they are connected to a pixel with intensity gradient above maxVal.

Hysteresis thresholding example on two lines

Edge A is above maxVal so is considered an edge. Edge B is in between maxVal and minVal but is not connected to any edge above maxVal so is discarded. Edge C is in between maxVal and minVal and is connected to edge A, an edge above maxVal, so is considered an edge.

For our pipeline, our frame is first grayscaled because we only need the luminance channel for detecting edges and a 5 by 5 gaussian blur is applied to decrease noise to

reduce false edges.

```python
# import cv2 as cv

def do_canny(frame):
    # Converts frame to grayscale because we only need the luminance channel for detecting edges
    gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
    # Applies a 5x5 gaussian blur with deviation of 0 to frame - not mandatory since Canny will
    blur = cv.GaussianBlur(gray, (5, 5), 0)
    # Applies Canny edge detector with minVal of 50 and maxVal of 150
    canny = cv.Canny(blur, 50, 150)
    return canny

# cap = cv.VideoCapture("input.mp4")
# while (cap.isOpened()):
#     ret, frame = cap.read()

    canny = do_canny(frame)

#     if cv.waitKey(10) & 0xFF == ord('q'):
#         break

# cap.release()
# cv.destroyAllWindows()
```

checkpoint2.py hosted with ♡ by GitHub view raw

4. Segmenting lane area

We will handcraft a triangular mask to segment the lane area and discard the irrelevant areas in the frame to increase the effectiveness of our later stages.

The triangular mask will be defined by three coordinates, indicated by the green circles.

```python
# import cv2 as cv
import numpy as np
import matplotlib.pyplot as plt

# def do_canny(frame):
#     gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
#     blur = cv.GaussianBlur(gray, (5, 5), 0)
#     canny = cv.Canny(blur, 50, 150)
#     return canny

def do_segment(frame):
    # Since an image is a multi-directional array containing the relative intensities of each pi
    # frame.shape[0] give us the number of rows of pixels the frame has. Since height begins fro
    height = frame.shape[0]
    # Creates a triangular polygon for the mask defined by three (x, y) coordinates
    polygons = np.array([
                            [(0, height), (800, height), (380, 290)]
                        ])
    # Creates an image filled with zero intensities with the same dimensions as the frame
    mask = np.zeros_like(frame)
    # Allows the mask to be filled with values of 1 and the other areas to be filled with values
    cv.fillPoly(mask, polygons, 255)
    # A bitwise and operation between the mask and frame keeps only the triangular area of the f
    segment = cv.bitwise_and(frame, mask)
    return segment

# cap = cv.VideoCapture("input.mp4")
# while (cap.isOpened()):
#     ret, frame = cap.read()
    # canny = do_canny(frame)

    # First, visualize the frame to figure out the three coordinates defining the triangular mas
    plt.imshow(frame)
    plt.show()
    segment = do_segment(canny)

#     if cv.waitKey(10) & 0xFF == ord('q'):
#         break

# cap.release()
# cv.destroyAllWindows()
```

5. Hough transform

In the Cartesian coordinate system, we can represent a straight line as `y = mx + b` by plotting y against x. However, we can also represent this line as a single point in Hough space by plotting b against m. For example, a line with the equation `y = 2x + 1` may be represented as `(2, 1)` in Hough space.

Now, what if instead of a line, we had to plot a point in the Cartesian coordinate system. There are many possible lines which can pass through this point, each line with different values for parameters m and b. For example, a point at (2, 12) can be passed by `y = 2x + 8`, `y = 3x + 6`, `y = 4x + 4`, `y = 5x + 2`, `y = 6x`, and so on. These possible lines can be plotted in Hough space as `(2, 8)`, `(3, 6)`, `(4, 4)`, `(5, 2)`, `(6, 0)`. Notice that this produces a line of m against b coordinates in Hough space.

Whenever we see a series of points in a Cartesian coordinate system and know that these points are connected by some line, we can find the equation of that line by first plotting each point in the Cartesian coordinate system to the corresponding line in Hough space, then finding the point of intersection in Hough space. The point of intersection in Hough space represents the m and b values that pass consistently through all of the points in the series.

Since our frame passed through the Canny Detector may be interpreted simply as a series of white points representing the edges in our image space, we can apply the same technique to identify which of these points are connected to the same line, and if they are connected, what its equation is so that we can plot this line on our frame.

For the simplicity of explanation, we used Cartesian coordinates to correspond to Hough space. However, there is one mathematical flaw with this approach: When the line is vertical, the gradient is infinity and cannot be represented in Hough space. To solve this problem, we will use Polar coordinates instead. The process is still the same just that other than plotting m against b in Hough space, we will be plotting r against θ.

For example, for the points on the Polar coordinate system with `x = 8` and `y = 6`, `x = 4` and `y = 9`, `x = 12` and `y = 3`, we can plot the corresponding Hough space.

We see that the lines in Hough space intersect at `θ = 0.925` and `r = 9.6`. Since a line in the Polar coordinate system is given by `r = xcosθ + ysinθ`, we can induce that a single line crossing through all these points is defined as `9.6 = xcos0.925 + ysin0.925`.

Generally, the more curves intersecting in Hough space means that the line represented by that intersection corresponds to more points. For our implementation, we will define a minimum threshold number of intersections in Hough space to detect a line. Therefore, Hough transform basically keeps track of the Hough space intersections of every point in the frame. If the number of intersections exceeds a defined threshold, we identify a line with the corresponding θ and r parameters.

We apply Hough Transform to identify two straight lines — which will be our left and right lane boundaries

```
# import cv2 as cv
# import numpy as np
# # import matplotlib.pyplot as plt

# def do_canny(frame):
#     gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
#     blur = cv.GaussianBlur(gray, (5, 5), 0)
#     canny = cv.Canny(blur, 50, 150)
#     return canny

# def do_segment(frame):
#     height = frame.shape[0]
#     polygons = np.array([
#                            [(0, height), (800, height), (380, 290)]
#                        ])
#     mask = np.zeros_like(frame)
#     cv.fillPoly(mask, polygons, 255)
#     segment = cv.bitwise_and(frame, mask)
#     return segment

# cap = cv.VideoCapture("input.mp4")
# while (cap.isOpened()):
#     ret, frame = cap.read()
#     canny = do_canny(frame)
#     # plt.imshow(frame)
#     # plt.show()
#     segment = do_segment(canny)

    # cv.HoughLinesP(frame, distance resolution of accumulator in pixels (larger = less precisio
    hough = cv.HoughLinesP(segment, 2, np.pi / 180, 100, np.array([]), minLineLength = 100, maxL

#     if cv.waitKey(10) & 0xFF == ord('q'):
#         break

# cap.release()
# cv.destroyAllWindows()
```

checkpoint4.py hosted with ♡ by GitHub view raw

6. Visualization

The lane is visualized as two light green, linearly fitted polynomials which will be overlayed on our input frame.

```
# import cv2 as cv
# import numpy as np
# # import matplotlib.pyplot as plt

# def do_canny(frame):
#     gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
#     blur = cv.GaussianBlur(gray, (5, 5), 0)
#     canny = cv.Canny(blur, 50, 150)
#     return canny

# def do_segment(frame):
#     height = frame.shape[0]
#     polygons = np.array([
#                             [(0, height), (800, height), (380, 290)]
#                         ])
#     mask = np.zeros_like(frame)
#     cv.fillPoly(mask, polygons, 255)
#     segment = cv.bitwise_and(frame, mask)
#     return segment

def calculate_lines(frame, lines):
    # Empty arrays to store the coordinates of the left and right lines
    left = []
    right = []
    # Loops through every detected line
    for line in lines:
        # Reshapes line from 2D array to 1D array
        x1, y1, x2, y2 = line.reshape(4)
        # Fits a linear polynomial to the x and y coordinates and returns a vector of coefficien
        parameters = np.polyfit((x1, x2), (y1, y2), 1)
        slope = parameters[0]
        y_intercept = parameters[1]
        # If slope is negative, the line is to the left of the lane, and otherwise, the line is
        if slope < 0:
            left.append((slope, y_intercept))
        else:
            right.append((slope, y_intercept))
    # Averages out all the values for left and right into a single slope and y-intercept value f
    left_avg = np.average(left, axis = 0)
    right_avg = np.average(right, axis = 0)
    # Calculates the x1, y1, x2, y2 coordinates for the left and right lines
```

```
    # Calculates the x1, y1, x2, y2 coordinates for the left and right lines
    left_line = calculate_coordinates(frame, left_avg)
    right_line = calculate_coordinates(frame, right_avg)
    return np.array([left_line, right_line])

def calculate_coordinates(frame, parameters):
    slope, intercept = parameters
    # Sets initial y-coordinate as height from top down (bottom of the frame)
    y1 = frame.shape[0]
    # Sets final y-coordinate as 150 above the bottom of the frame
    y2 = int(y1 - 150)
    # Sets initial x-coordinate as (y1 - b) / m since y1 = mx1 + b
    x1 = int((y1 - intercept) / slope)
    # Sets final x-coordinate as (y2 - b) / m since y2 = mx2 + b
    x2 = int((y2 - intercept) / slope)
    return np.array([x1, y1, x2, y2])

def visualize_lines(frame, lines):
    # Creates an image filled with zero intensities with the same dimensions as the frame
    lines_visualize = np.zeros_like(frame)
    # Checks if any lines are detected
    if lines is not None:
        for x1, y1, x2, y2 in lines:
            # Draws lines between two coordinates with green color and 5 thickness
            cv.line(lines_visualize, (x1, y1), (x2, y2), (0, 255, 0), 5)
    return lines_visualize

# cap = cv.VideoCapture("input.mp4")
# while (cap.isOpened()):
#     ret, frame = cap.read()
#     canny = do_canny(frame)
#     # plt.imshow(frame)
#     # plt.show()
#     segment = do_segment(canny)
#     hough = cv.HoughLinesP(segment, 2, np.pi / 180, 100, np.array([]), minLineLength = 100, ma

    # Averages multiple detected lines from hough into one line for left border of lane and one
    lines = calculate_lines(frame, hough)
    # Visualizes the lines
    lines_visualize = visualize_lines(frame, lines)
    # Overlays lines on frame by taking their weighted sums and adding an arbitrary scalar value
    output = cv.addWeighted(frame, 0.9, lines_visualize, 1, 1)
    # Opens a new window and displays the output frame
    cv.imshow("output", output)

```

```
#     if cv.waitKey(10) & 0xFF == ord('q'):
#         break

# cap.release()
# cv.destroyAllWindows()
```

Now, open Terminal and run `python detector.py` to test your simple lane detector! In case you have missed any code, here is the full solution with comments:

```
import cv2 as cv
import numpy as np
# import matplotlib.pyplot as plt

def do_canny(frame):
    # Converts frame to grayscale because we only need the luminance channel for detecting edges
    gray = cv.cvtColor(frame, cv.COLOR_RGB2GRAY)
    # Applies a 5x5 gaussian blur with deviation of 0 to frame - not mandatory since Canny will
    blur = cv.GaussianBlur(gray, (5, 5), 0)
    # Applies Canny edge detector with minVal of 50 and maxVal of 150
    canny = cv.Canny(blur, 50, 150)
    return canny

def do_segment(frame):
    # Since an image is a multi-directional array containing the relative intensities of each p
    # frame.shape[0] give us the number of rows of pixels the frame has. Since height begins fr
    height = frame.shape[0]
    # Creates a triangular polygon for the mask defined by three (x, y) coordinates
    polygons = np.array([
                            [(0, height), (800, height), (380, 290)]
                        ])
    # Creates an image filled with zero intensities with the same dimensions as the frame
    mask = np.zeros_like(frame)
    # Allows the mask to be filled with values of 1 and the other areas to be filled with values
    cv.fillPoly(mask, polygons, 255)
    # A bitwise and operation between the mask and frame keeps only the triangular area of the
    segment = cv.bitwise_and(frame, mask)
    return segment

def calculate_lines(frame, lines):
    # Empty arrays to store the coordinates of the left and right lines
    left = []
    right = []
```

```
    # Loops through every detected line
    for line in lines:
        # Reshapes line from 2D array to 1D array
        x1, y1, x2, y2 = line.reshape(4)
        # Fits a linear polynomial to the x and y coordinates and returns a vector of coefficie
        parameters = np.polyfit((x1, x2), (y1, y2), 1)
        slope = parameters[0]
        y_intercept = parameters[1]
        # If slope is negative, the line is to the left of the lane, and otherwise, the line is
        if slope < 0:
            left.append((slope, y_intercept))
        else:
            right.append((slope, y_intercept))
    # Averages out all the values for left and right into a single slope and y-intercept value
    left_avg = np.average(left, axis = 0)
    right_avg = np.average(right, axis = 0)
    # Calculates the x1, y1, x2, y2 coordinates for the left and right lines
    left_line = calculate_coordinates(frame, left_avg)
    right_line = calculate_coordinates(frame, right_avg)
    return np.array([left_line, right_line])

def calculate_coordinates(frame, parameters):
    slope, intercept = parameters
    # Sets initial y-coordinate as height from top down (bottom of the frame)
    y1 = frame.shape[0]
    # Sets final y-coordinate as 150 above the bottom of the frame
    y2 = int(y1 - 150)
    # Sets initial x-coordinate as (y1 - b) / m since y1 = mx1 + b
    x1 = int((y1 - intercept) / slope)
    # Sets final x-coordinate as (y2 - b) / m since y2 = mx2 + b
    x2 = int((y2 - intercept) / slope)
    return np.array([x1, y1, x2, y2])

def visualize_lines(frame, lines):
    # Creates an image filled with zero intensities with the same dimensions as the frame
    lines_visualize = np.zeros_like(frame)
    # Checks if any lines are detected
    if lines is not None:
        for x1, y1, x2, y2 in lines:
            # Draws lines between two coordinates with green color and 5 thickness
            cv.line(lines_visualize, (x1, y1), (x2, y2), (0, 255, 0), 5)
    return lines_visualize

# The video feed is read in as a VideoCapture object
cap = cv.VideoCapture("input.mp4")
```

```
while (cap.isOpened()):
    # ret = a boolean return value from getting the frame, frame = the current frame being proje
    ret, frame = cap.read()
    canny = do_canny(frame)
    cv.imshow("canny", canny)
    # plt.imshow(frame)
    # plt.show()
    segment = do_segment(canny)
    hough = cv.HoughLinesP(segment, 2, np.pi / 180, 100, np.array([]), minLineLength = 100, max
    # Averages multiple detected lines from hough into one line for left border of lane and one
    lines = calculate_lines(frame, hough)
    # Visualizes the lines
    lines_visualize = visualize_lines(frame, lines)
    cv.imshow("hough", lines_visualize)
    # Overlays lines on frame by taking their weighted sums and adding an arbitrary scalar valu
    output = cv.addWeighted(frame, 0.9, lines_visualize, 1, 1)
    # Opens a new window and displays the output frame
    cv.imshow("output", output)
    # Frames are read by intervals of 10 milliseconds. The programs breaks out of the while loo
    if cv.waitKey(10) & 0xFF == ord('q'):
        break
# The following frees up resources and closes all windows
cap.release()
cv.destroyAllWindows()
```

Approach 2: Spatial CNN

This rather handcrafted traditional method in Approach 1 seems to work decently… at least for clear straight roads. However, it is fairly obvious that this method would break instantly on curved lanes or sharp turns. Also, we noticed that the presence of markings consisting of straight lines on the lanes, such as painted arrow signs, may confuse the lane detector from time to time, evident from the glitches in the demo rendering. One way to overcome this may be to further refine the triangular mask into two separate, more precise masks. Nonetheless, these rather arbitrary mask parameters simply cannot adapt to various changing road environments. Another shortcoming is that lanes with dotted markings or with no clear markings at all are also ignored by the lane detector since there are no continuous straight lines that satisfy the Hough transform threshold. Finally, weather and lighting conditions affecting the visibility of the lines may also be an issue.

1. Architecture

While Convolutional Neural Networks (CNNs) have proven to be effective architectures for both recognizing simple features at lower layers of images (e.g. edges, color gradients) as well as complex features and entities in deeper levels (e.g. object recognition), they often struggle to represent the "pose" of these features and entities — that is, CNNs are great for extracting semantics from raw pixels but perform poorly on capturing the spatial relationships (e.g. rotational and translational relationships) of pixels in a frame. These spatial relationships, however, are important for the task of lane detection, where there are strong shape priors but weak appearance coherences.

For example, it is hard to determine traffic poles solely by extracting semantic features as they lack distinct and coherent appearance cues and are often occluded.

The car to the right of the top left image and the motorbike to the right of the bottom left image occlude the right lane markings and negatively affect CNN results

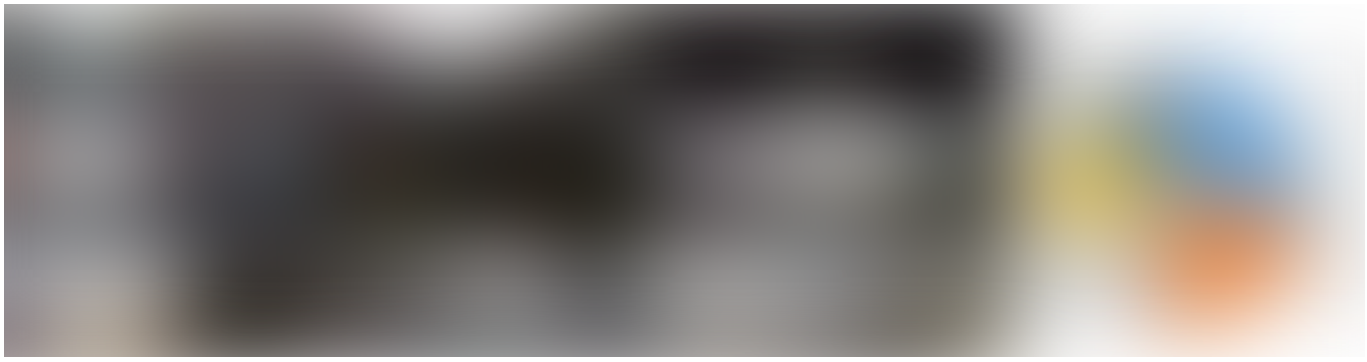
However, since we know traffic poles usually exhibit similar spatial relationships such as to stand vertically and are placed alongside the left and right of roads, we see the importance of reinforcing spatial information. A similar case follows for detecting lanes.

To address this issue, Spatial CNN (SCNN) proposes an architecture which "generalizes traditional deep layer-by-layer convolutions to slice-by slice convolutions within feature maps". What does this mean? In a traditional layer-by-layer CNN, each convolution layer receives input from its preceding layer, applies convolutions and nonlinear activation, and sends the output to the succeeding layer. SCNN takes this a step further by treating individual feature map rows and columns as the "layers", applying the same process sequentially (where sequentially means that a slice passes information to the succeeding slice only after it has received information from the preceding slices), allowing message

passing of pixel information between neurons within the same layer, effectively increasing emphasis on spatial information.

SCNN is relatively new, published only earlier this year (2018), but have already outperformed the likes of ReNet (RNN), MRFNet (MRF+CNN), much deeper ResNet architectures, and placed first on the TuSimple Benchmark Lane Detection Challenge with 96.53% accuracy.

In addition, alongside the publication of SCNN, the authors also released CULane Dataset, a large scale dataset with annotations of traffic lanes with cubic spines. CULane Dataset also contains many challenging scenarios, including occlusions and varying lighting conditions.



2. Model

Lane detection requires precise pixel-wise identification and prediction of lane curves. Instead of training for lane presence directly and performing clustering afterwards, the authors of SCNN treated the blue, green, red, and yellow lane markings as four separate classes. The model outputs probability maps (probmaps) for each curve, similar to semantic segmentation tasks, then passes the probmaps through a small network to predict the final cubic spines. The model is based on the DeepLab-LargeFOV model variant.

For each lane marking with over 0.5 existence value, the corresponding probmap is searched by 20 row intervals for the position with the highest response. To determine whether if a lane marking is detected, the Intersection-over-Union (IoU) between the ground truth (correct labels) and prediction is calculated, where IoUs above a set threshold are evaluated as true positives (TP) to calculate precision and recall.

3. Testing and Training

You can follow this repository to reproduce the results in the SCNN paper or test your own model with the CULane Dataset.

And that's it! 🎉 Hopefully this tutorial showed you how to build a simple lane detector using the traditional approach which involves many handcrafted features and fine-tuning, and also introduced you to an alternative method which follows the recent trend of solving almost any type of computer vision problem: you can add a convolutional neural network to that!

Hats off to you for completing this tutorial and I hope you enjoyed it 📓*. Feel free to follow for more upcoming tutorials! :)*