Jetson racer autonomous car tutorial

This tutorial explains how to create a simple autonomous RC car based on jet-racer RC car kit and Jetson Nano (NVIDIA microprocessor).

Note: most of the specific information and the full instructions are in the mentioned links.

Step 1

Setup the jetson-nano

(Install the jetson-nano developer kit image by using SD card)

Jetson Nano is a small, powerful computer for embedded applications and AI IoT that delivers the power of modern AI.

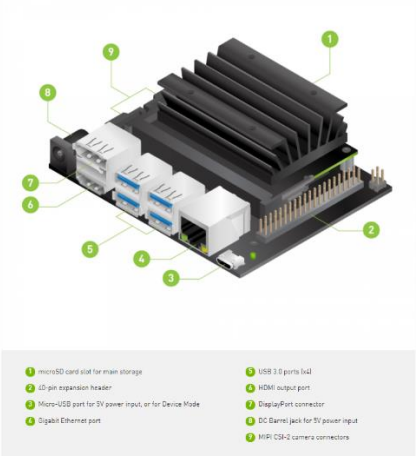
To start working with jetson-nano, first complete the "Software Setup" (download the operation system image and complete the instructions).

this link contains all the necessary instructions:

https://github.com/waveshare/jetracer/blob/master/docs/software_setup.md

this video is simply explaining this step:

https://www.youtube.com/watch?v=uvU8AXY1170



Step 2

Assemble the jet racer kit

Jet-Racer is an autonomous AI racing car Powered by Nvidia Jetson Nano.

this link contains the complete jet-racer assemble manual:

https://www.waveshare.com/wiki/JetRacer_Assembly_Manual

this video is simply explaining this step:

https://www.youtube.com/watch?v=Mn2QYPFADdo

Step 3

Controlling the car by using python interface

To start working with the jet-racer car it is necessary to setup the "jet-racer" lib (enable python interface to manage the car controllers).

Here is the link to the jet-racer library repository:

https://github.com/waveshare/jetracer

after the running the setup, it is recommended to try basic motion test.

the repository contains simple code examples.

Execute the following block of code by selecting it and clicking `ctrl + enter` to create an `NvidiaRacecar` class.

In [ ]:
```
from jetracer.nvidia_racecar import NvidiaRacecar
car = NvidiaRacecar()
```

The `NvidiaRacecar` implements the `Racecar` class, so it has two attributes `throttle` and `steering`.

We can assign values in the range `[-1, 1]` to these attributes. Execute the following to set the steering to 0.4.

> If the car does not respond, it may still be in `manual` mode. Flip the manual override switch on the RC transmitter.

In [ ]:
```
car.steering = 0.3
```

The `NvidiaRacecar` class has two values `steering_gain` and `steering_bias` that can be used to calibrate the steering.

We can view the default values by executing the cells below.

In [ ]:
```
print(car.steering_gain)
```

In [ ]:
```
print(car.steering_offset)
```

The final steering value is computed using the equation

$y = a \times x + b$

Where,

- a is `car.steering_gain`
- b is `car.steering_offset`
- x is `car.steering`
- y is the value written to the motor driver

You can adjust these values calibrate the car so that setting a value of `0` moves forward, and setting a value of `1` goes fully right, and `-1` fully left.

To set the throttle of the car to `0.2`, you can call the following.

> Give JetRacer lots of space to move, and be ready on the manual override, JetRacer is *fast*

In [ ]:
```
car.throttle = 0.0
```

The throttle also has a gain value that could be used to control the speed response. The throttle output is computed as

$y = a \times x$

Where,

- a is `car.throttle_gain`
- x is `car.throttle`
- y is the value written to the speed controller

Execute the following to print the default gain

In [ ]:
```
print(car.throttle_gain)
```

Set the following to limit the throttle to half

In [ ]:
```
car.throttle_gain = 0.5
```

Please note the throttle is directly mapped to the RC car. When the car is stopped and a negative throttle is set, it will reverse. If the car is moving forward and a negative throttle is set, it will brake.

Step 4

Using deep learning for objects detection

The car system must be able to recognize objects (cars, traffic lines and pedestrians) for making the correct decisions if there is a danger on the road.

Code your own Python program for object detection using Jetson Nano and deep learning, then experiment with real time detection on a live camera stream.

Follow Jetson-Inference instructional guide (to enable deep-learning include object detection).

Here is the link to jetson inference repository:

https://github.com/dusty-nv/jetson-inference/

object detection inference video:

https://www.youtube.com/watch?v=obt60r8ZeB0&list=PL5B692fm6--uQRRDTPsJDp4o0xbzkoyf8&index=13

training object detection models video:

https://www.youtube.com/watch?v=2XMkPW_sIGg&list=PL5B692fm6--uQRRDTPsJDp4o0xbzkoyf8&index=14

Step 5

Using Canny-Hough with openCV for lane detection

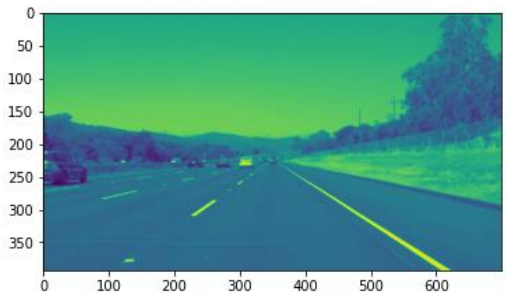
We are using canny detector-Hough transform based lane detection by tracking road lanes using computer vision techniques.

1)

Canny Edge detector needs grey scale images, hence we need to convert our image into grey scale. We are collapsing 3 channels of pixel value (Red, Green, and Blue) into a single channel with a pixel value range of [0,255].

```
# Converting color image to grayscale image
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

plt.imshow(gray)
plt.show()
```



2)

Creating a Gaussian blur over our grey scale image.

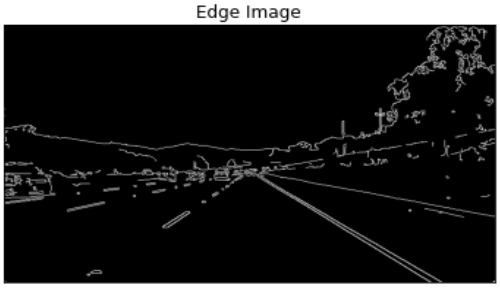
```
blur = cv2.GaussianBlur(gray, (5, 5), 0)
plt.imshow(blur,cmap='gray')
plt.title('GaussianBlur'), plt.xticks([]), plt.yticks([])
plt.show()
```

3)

edge detection.

```
edges = cv2.Canny(img,100,200)
plt.imshow(edges,cmap='gray')
plt.title('Edge Image'), plt.xticks([]), plt.yticks([])
plt.show()
```

Edge Image



4)

apply a mask to the original RGB image to return the pixels we are interested in.

```
img_hsv = cv2.cvtColor(img, cv2.COLOR_RGB2HSV)
lower_yellow = np.array([20, 100, 100], dtype = "uint8")
upper_yellow = np.array([30, 255, 255], dtype="uint8")

mask_yellow = cv2.inRange(img_hsv, lower_yellow, upper_yellow)
mask_white = cv2.inRange(gray, 200, 255)
mask_yw = cv2.bitwise_or(mask_white, mask_yellow)
mask_yw_image = cv2.bitwise_and(gray, mask_yw)
```

```
plt.imshow(mask_yw_image,cmap='gray')
plt.title('mask_yw_image'), plt.xticks([]), plt.yticks([])
plt.show()
```

maskyellow

5)

Now that we've defined all the edges in the image, we need to isolate the edges that correspond with the lane lines. Here's how we're going to do that.

```
def region(image):
    height, width = image.shape
    triangle = np.array([
                        [(100, height), (475, 325), (width, height)]
                        ])

    mask = np.zeros_like(image)

    mask = cv2.fillPoly(mask, triangle, 255)
    mask = cv2.bitwise_and(image, mask)
    return mask
```

This function will isolate a certain hard-coded region in the image where the lane lines are. It takes one parameter, the Canny image and outputs the isolated region.

6)

Hough line transform, the part that turns those clusters of white pixels from our isolated region into actual lines.

```
lines = cv2.HoughLinesP(isolated, rho=2, theta=np.pi/180,
threshold=100, np.array([]), minLineLength=40, maxLineGap=5)
```

7)

Optimizing and Displaying lines.

To average the lines, we're going to define a function called "average".

```
def average(image, lines):
    left = []
    right = []
    for line in lines:
        print(line)
        x1, y1, x2, y2 = line.reshape(4)
        parameters = np.polyfit((x1, x2), (y1, y2), 1)
        slope = parameters[0]
        y_int = parameters[1]
        if slope < 0:
            left.append((slope, y_int))
        else:
            right.append((slope, y_int))
```

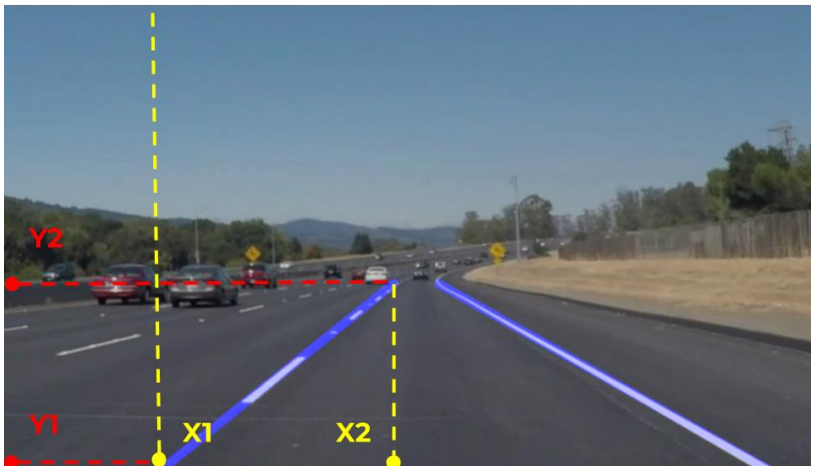
This function averages out the lines made in the *cv2.HoughLinesP* function. It will find the average slope and y-intercept of the line segments on the left and the right and output two solid lines instead (one on the left and other on the right).

Next, we must take the average of the slopes and y-intercepts from both lists.

```
    right_avg = np.average(right, axis=0)
    left_avg = np.average(left, axis=0)
    left_line = make_points(image, left_avg)
    right_line = make_points(image, right_avg)
    return np.array([left_line, right_line])
```

Now that we have the average slope and y-intercept for both lists, let's define the start and endpoints for both lists.

```
def make_points(image, average):
 slope, y_int = average
 y1 = image.shape[0]
 y2 = int(y1 * (3/5))
 x1 = int((y1 - y_int) // slope)
 x2 = int((y2 - y_int) // slope)
 return np.array([x1, y1, x2, y2])
```



Next, we're going to create a function which takes these points and makes lines out of them.

```
def display_lines(image, lines):
 lines_image = np.zeros_like(image)
 if lines is not None:
   for line in lines:
     x1, y1, x2, y2 = line
     cv2.line(lines_image, (x1, y1), (x2, y2), (255, 0, 0), 10)
 return lines_image
```

The following diagram is an overview of our pipe line.

Over view of Canny-Hough detection system

Here is the link to the complete lane detection tutorial:

https://medium.com/analytics-vidhya/building-a-lane-detection-system-f7a727c6694

link to our autonomous car project GitHub repository:

https://github.com/DorGetter/FinalProject