Detect and Recognize Car License Plate from a video in real time

Recognizing Car License Plate is a very important task for a camera surveillance-based security system. We can extract the license plate from an image using some computer vision techniques and then we can use Optical Character Recognition to recognize the license number. Here I will guide you through the whole procedure of this task.

**Requirements:**

*opencv-python 3.4.2  
numpy 1.17.2  
skimage 0.16.2  
tensorflow 1.15.0  
imutils 0.5.3*

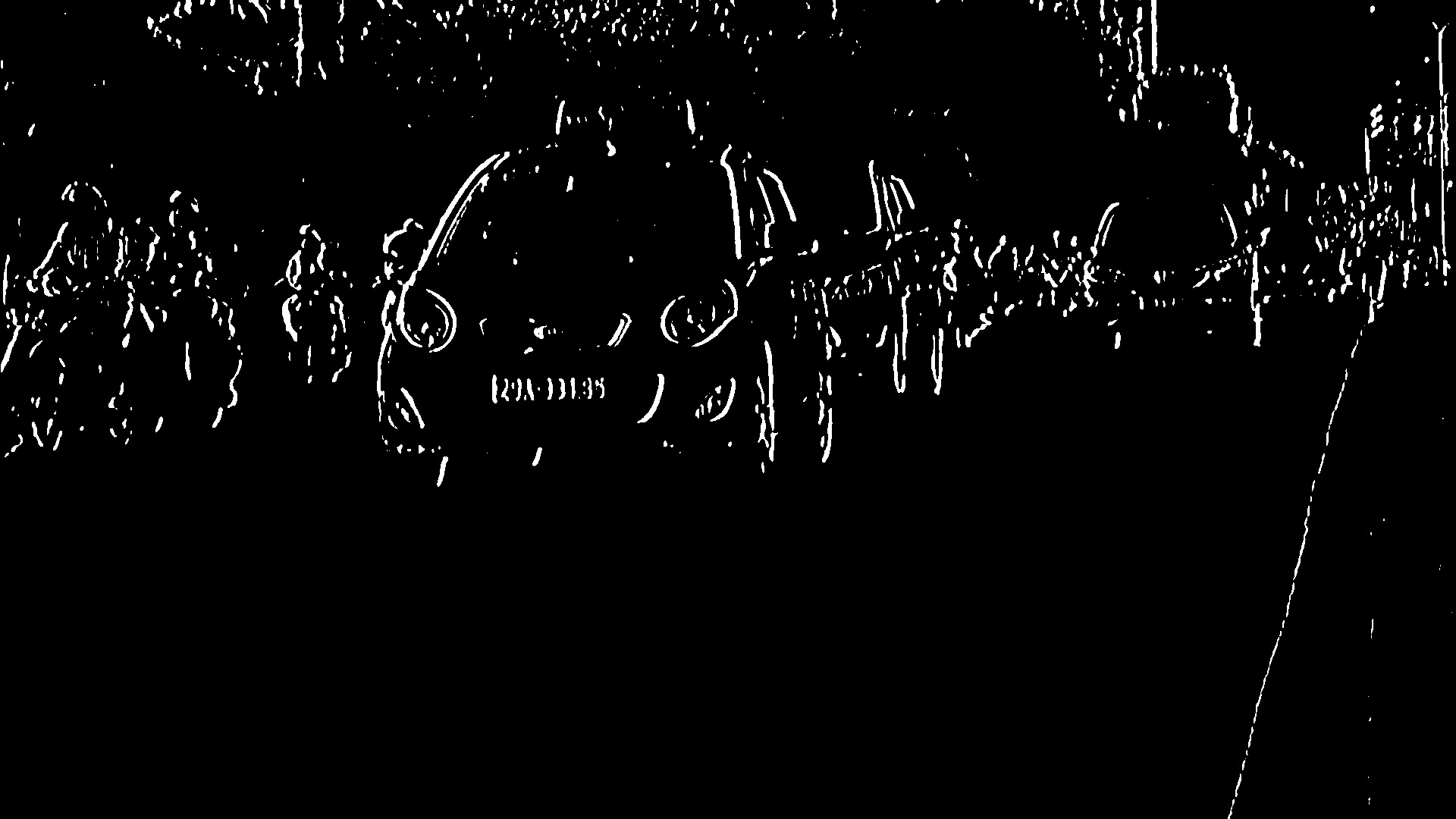
**Example:**

***Input:*** *****Output:*** *29A33185  
https://media.geeksforgeeks.org/wp-content/uploads/20200326013049/plate.jpg*

**Approach:**

* Find all the contours in the image.
* Find the bounding rectangle of every contour.
* Compare and validate the sides ratio and area of every bounding rectangle with an average license plate.
* Apply image segmentation in the image inside validated contour to find characters in it.
* Recognize characters using an OCR.

**Methodology:**

1. To reduce the noise we need to blur the input Image with [Gaussian Blur](https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html) then convert the it to grayscale.  
   
2. Find vertical edges in the image.  
   
3. To reveal the plate we have to binarize the image. For this apply Otsu’s Thresholding on the vertical edge image. In other thresholding methods we have to choose a threshold value to binarize the image but Otsu’s Thresholding determines the value automatically.  
   
4. Apply Closing Morphological Transformation on thresholded image. Closing is useful to fill small black regions between white regions in a thresholded image. It reveals the rectangular white box of license plate.  
   

Above 4 steps are performed by **preprocess** method of class **PlateFinder**

|  |
| --- |
| def preprocess(self, input\_img):        imgBlurred = cv2.GaussianBlur(input\_img, (7, 7), 0)        # convert to gray      gray = cv2.cvtColor(imgBlurred,                          cv2.COLOR\_BGR2GRAY)        # sobelX to get the vertical edges      sobelx = cv2.Sobel(gray, cv2.CV\_8U,                         1, 0, ksize = 3)         # otsu's thresholding      ret2, threshold\_img = cv2.threshold(sobelx,0, 255,                             cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)        element = self.element\_structure      morph\_n\_thresholded\_img = threshold\_img.copy()      cv2.morphologyEx(src = threshold\_img,                       op = cv2.MORPH\_CLOSE,                       kernel = element,                       dst = morph\_n\_thresholded\_img)        return morph\_n\_thresholded\_img |

1. To detect the plate we need to find contours in the image. It is important to binarize and morph the image before finding contours so that it can find more relevant and less number of contours in the image. If you draw all the extracted contours on original image, it would look like this:  
   

This step is performed by **extract\_contours** method of class **PlateFinder**

|  |
| --- |
| def extract\_contours(self, after\_preprocess):        \_, contours, \_ = cv2.findContours(after\_preprocess,                                        mode = cv2.RETR\_EXTERNAL,                                        method = cv2.CHAIN\_APPROX\_NONE)      return contours |

1. Now find the [minimum area rectangle](https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=minarearect#minarearect) enclosed by each of the contour and validate their side ratios and area. We have defined the minimum and maximum area of the plate as 4500 and 30000 respectively.

**Code:**Methods validating the area and side ratios of minimum area rectangle are **validateRatio** and  
**preRatioCheck** of class **PlateFinder**:

|  |
| --- |
| def validateRatio(self, rect):        (x, y), (width, height), rect\_angle = rect        if (width > height):          angle = -rect\_angle        else:          angle = 90 + rect\_angle        if angle > 15:          return False        if (height == 0 or width == 0):          return False        area = width \* height        if not self.preRatioCheck(area, width, height):          return False        else:          return True    def preRatioCheck(self, area, width, height):        min = self.min\_area      max = self.max\_area        ratioMin = 2.5      ratioMax = 7        ratio = float(width) / float(height)        if ratio < 1:          ratio = 1 / ratio        if (area < min or area > max) or (ratio < ratioMin or ratio > ratioMax):          return False        return True |

1. Now find the contours in the validated region and validate the side ratios and area of the bounding rectangle of the largest contour in that region. After validating you will get a perfect contour of a license plate. Now extract that contour from the original image. You will get the image of plate:  
   https://media.geeksforgeeks.org/wp-content/uploads/20200326013049/plate.jpg
2. **Code:**This step is performed by **clean\_plate** and **ratioCheck** method of class **PlateFinder**.

|  |
| --- |
| def clean\_plate(self, plate):        gray = cv2.cvtColor(plate, cv2.COLOR\_BGR2GRAY)      thresh = cv2.adaptiveThreshold(gray, 255,                                     cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,                                     cv2.THRESH\_BINARY, 11, 2)        \_, contours, \_ = cv2.findContours(thresh.copy(),                                        cv2.RETR\_EXTERNAL,                                        cv2.CHAIN\_APPROX\_NONE)        if contours:            areas = [cv2.contourArea(c) for c in contours]            # index of the largest contour in the          # areas array          max\_index = np.argmax(areas)            max\_cnt = contours[max\_index]          max\_cntArea = areas[max\_index]          x, y, w, h = cv2.boundingRect(max\_cnt)            if not self.ratioCheck(max\_cntArea,                                 plate.shape[1],                                 plate.shape[0]):              return plate, False, None            return plate, True, [x, y, w, h]        else:          return plate, False, None    def ratioCheck(self, area, width, height):        min = self.min\_area      max = self.max\_area        ratioMin = 3      ratioMax = 6        ratio = float(width) / float(height)        if ratio < 1:          ratio = 1 / ratio        if (area < min or area > max) or (ratio < ratioMin or ratio > ratioMax):          return False        return True |

1. To recognize the characters on license plate precisely, we have to apply image segmentation. For that first step is to extract the value channel from the HSV format of the plate’s image. It would look like-

https://media.geeksforgeeks.org/wp-content/uploads/20200326013712/value2.jpg

1. Now apply adaptive thresholding on the plate’s value channel image to binarize it and reveal the characters. The image of plate can have different lightning conditions in different areas, in that case adaptive thresholding can be more suitable to binarize because it uses different threshold values for different regions based on the brightness of the pixels in the region around it.

https://media.geeksforgeeks.org/wp-content/uploads/20200326015459/after_adaptive_thresholding.jpg

1. After binarizing apply bitwise not operation on the image to find the connected components in the image so that we can extract character candidates.

https://media.geeksforgeeks.org/wp-content/uploads/20200326020326/adaptive_not.jpg

1. Construct a mask to display all the character components and then find contours in mask. After extracting the contours take the largest one, find its bounding rectangle and validate side ratios.
2. After validating the side ratios find the convex hull of the contour ad draw it on the character candidate mask. The mask would look like-



1. Now find all the contours in the character candidate mask and extract those contour areas from the plate’s value thresholded image, you will get all the characters separately.  
     
   Steps **8** to **13** are performed by **segment\_chars** function that you can find below in the full source code. The driver code for the functions used in steps 6 to 13 is written in the method **check\_plate** of class **PlateFinder**.
2. Now use OCR to recognize the character one by one.

**Full Source Code with its working:** First, create a **PlateFinder** class that finds the license plates and validates its size ratio and area.

|  |
| --- |
| import cv2  import numpy as np  from skimage.filters import threshold\_local  import tensorflow as tf  from skimage import measure  import imutils      def sort\_cont(character\_contours):      """      To sort contours      """      i = 0      boundingBoxes = [cv2.boundingRect(c) for c in character\_contours]        (character\_contours, boundingBoxes) = zip(\*sorted(zip(character\_contours,                                                            boundingBoxes),                                                        key = lambda b: b[1][i],                                                        reverse = False))        return character\_contours      def segment\_chars(plate\_img, fixed\_width):        """      extract Value channel from the HSV format      of image and apply adaptive thresholding      to reveal the characters on the license plate      """      V = cv2.split(cv2.cvtColor(plate\_img, cv2.COLOR\_BGR2HSV))[2]        thresh = cv2.adaptiveThreshold(value, 255,                                     cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,                                     cv2.THRESH\_BINARY,                                     11, 2)        thresh = cv2.bitwise\_not(thresh)        # resize the license plate region to      # a canoncial size      plate\_img = imutils.resize(plate\_img, width = fixed\_width)      thresh = imutils.resize(thresh, width = fixed\_width)      bgr\_thresh = cv2.cvtColor(thresh, cv2.COLOR\_GRAY2BGR)        # perform a connected components analysis      # and initialize the mask to store the locations      # of the character candidates      labels = measure.label(thresh, neighbors = 8, background = 0)        charCandidates = np.zeros(thresh.shape, dtype ='uint8')        # loop over the unique components      characters = []      for label in np.unique(labels):            # if this is the background label, ignore it          if label == 0:              continue          # otherwise, construct the label mask to display          # only connected components for the current label,          # then find contours in the label mask          labelMask = np.zeros(thresh.shape, dtype ='uint8')          labelMask[labels == label] = 255            cnts = cv2.findContours(labelMask,                       cv2.RETR\_EXTERNAL,                       cv2.CHAIN\_APPROX\_SIMPLE)            cnts = cnts[0] if imutils.is\_cv2() else cnts[1]            # ensure at least one contour was found in the mask          if len(cnts) > 0:                # grab the largest contour which corresponds              # to the component in the mask, then grab the              # bounding box for the contour              c = max(cnts, key = cv2.contourArea)              (boxX, boxY, boxW, boxH) = cv2.boundingRect(c)                # compute the aspect ratio, solodity, and              # height ration for the component              aspectRatio = boxW / float(boxH)              solidity = cv2.contourArea(c) / float(boxW \* boxH)              heightRatio = boxH / float(plate\_img.shape[0])                # determine if the aspect ratio, solidity,              # and height of the contour pass the rules              # tests              keepAspectRatio = aspectRatio < 1.0              keepSolidity = solidity > 0.15              keepHeight = heightRatio > 0.5 and heightRatio < 0.95                # check to see if the component passes              # all the tests              if keepAspectRatio and keepSolidity and keepHeight and boxW > 14:                    # compute the convex hull of the contour                  # and draw it on the character candidates                  # mask                  hull = cv2.convexHull(c)                    cv2.drawContours(charCandidates, [hull], -1, 255, -1)        \_, contours, hier = cv2.findContours(charCandidates,                                           cv2.RETR\_EXTERNAL,                                           cv2.CHAIN\_APPROX\_SIMPLE)        if contours:          contours = sort\_cont(contours)            # value to be added to each dimension          # of the character          addPixel = 4          for c in contours:              (x, y, w, h) = cv2.boundingRect(c)              if y > addPixel:                  y = y - addPixel              else:                  y = 0              if x > addPixel:                  x = x - addPixel              else:                  x = 0              temp = bgr\_thresh[y:y + h + (addPixel \* 2),                                x:x + w + (addPixel \* 2)]                characters.append(temp)            return characters        else:          return None        class PlateFinder:      def \_\_init\_\_(self):            # minimum area of the plate          self.min\_area = 4500            # maximum area of the plate          self.max\_area = 30000            self.element\_structure = cv2.getStructuringElement(                                shape = cv2.MORPH\_RECT, ksize =(22, 3))        def preprocess(self, input\_img):            imgBlurred = cv2.GaussianBlur(input\_img, (7, 7), 0)            # convert to gray          gray = cv2.cvtColor(imgBlurred, cv2.COLOR\_BGR2GRAY)            # sobelX to get the vertical edges          sobelx = cv2.Sobel(gray, cv2.CV\_8U, 1, 0, ksize = 3)            # otsu's thresholding          ret2, threshold\_img = cv2.threshold(sobelx, 0, 255,                           cv2.THRESH\_BINARY + cv2.THRESH\_OTSU)            element = self.element\_structure          morph\_n\_thresholded\_img = threshold\_img.copy()          cv2.morphologyEx(src = threshold\_img,                           op = cv2.MORPH\_CLOSE,                           kernel = element,                           dst = morph\_n\_thresholded\_img)            return morph\_n\_thresholded\_img        def extract\_contours(self, after\_preprocess):            \_, contours, \_ = cv2.findContours(after\_preprocess,                                            mode = cv2.RETR\_EXTERNAL,                                            method = cv2.CHAIN\_APPROX\_NONE)          return contours        def clean\_plate(self, plate):            gray = cv2.cvtColor(plate, cv2.COLOR\_BGR2GRAY)          thresh = cv2.adaptiveThreshold(gray,                                         255,                                         cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,                                         cv2.THRESH\_BINARY,                                         11, 2)            \_, contours, \_ = cv2.findContours(thresh.copy(),                                            cv2.RETR\_EXTERNAL,                                            cv2.CHAIN\_APPROX\_NONE)            if contours:              areas = [cv2.contourArea(c) for c in contours]                # index of the largest contour in the area              # array              max\_index = np.argmax(areas)                max\_cnt = contours[max\_index]              max\_cntArea = areas[max\_index]              x, y, w, h = cv2.boundingRect(max\_cnt)              rect = cv2.minAreaRect(max\_cnt)                if not self.ratioCheck(max\_cntArea, plate.shape[1],                                                  plate.shape[0]):                  return plate, False, None                return plate, True, [x, y, w, h]            else:              return plate, False, None            def check\_plate(self, input\_img, contour):            min\_rect = cv2.minAreaRect(contour)            if self.validateRatio(min\_rect):              x, y, w, h = cv2.boundingRect(contour)              after\_validation\_img = input\_img[y:y + h, x:x + w]              after\_clean\_plate\_img, plateFound, coordinates = self.clean\_plate(                                                          after\_validation\_img)                if plateFound:                  characters\_on\_plate = self.find\_characters\_on\_plate(                                                after\_clean\_plate\_img)                    if (characters\_on\_plate is not None and len(characters\_on\_plate) == 8):                      x1, y1, w1, h1 = coordinates                      coordinates = x1 + x, y1 + y                      after\_check\_plate\_img = after\_clean\_plate\_img                        return after\_check\_plate\_img, characters\_on\_plate, coordinates            return None, None, None            def find\_possible\_plates(self, input\_img):            """          Finding all possible contours that can be plates          """          plates = []          self.char\_on\_plate = []          self.corresponding\_area = []            self.after\_preprocess = self.preprocess(input\_img)          possible\_plate\_contours = self.extract\_contours(self.after\_preprocess)            for cnts in possible\_plate\_contours:              plate, characters\_on\_plate, coordinates = self.check\_plate(input\_img, cnts)                if plate is not None:                  plates.append(plate)                  self.char\_on\_plate.append(characters\_on\_plate)                  self.corresponding\_area.append(coordinates)            if (len(plates) > 0):              return plates            else:              return None        def find\_characters\_on\_plate(self, plate):            charactersFound = segment\_chars(plate, 400)          if charactersFound:              return charactersFound        # PLATE FEATURES      def ratioCheck(self, area, width, height):            min = self.min\_area          max = self.max\_area            ratioMin = 3          ratioMax = 6            ratio = float(width) / float(height)            if ratio < 1:              ratio = 1 / ratio            if (area < min or area > max) or (ratio < ratioMin or ratio > ratioMax):              return False            return True        def preRatioCheck(self, area, width, height):            min = self.min\_area          max = self.max\_area            ratioMin = 2.5          ratioMax = 7            ratio = float(width) / float(height)            if ratio < 1:              ratio = 1 / ratio            if (area < min or area > max) or (ratio < ratioMin or ratio > ratioMax):              return False            return True        def validateRatio(self, rect):          (x, y), (width, height), rect\_angle = rect            if (width > height):              angle = -rect\_angle          else:              angle = 90 + rect\_angle            if angle > 15:              return False            if (height == 0 or width == 0):              return False            area = width \* height            if not self.preRatioCheck(area, width, height):              return False          else:              return True |

Here is the explanation of each and every method of **PlateFinder** class.  
In **preprocess** method, following step has been done:

* Blur the Image
* Convert to Grayscale
* Find vertical edges
* Threshold the vertical edged image.
* Close Morph the Threshold image.

Method **extract\_contours** returns all external contours from the preprocessed image.  
Method **find\_possible\_plates** precprocess the image with **preprocess** method then extracts contours by **extract\_contours** method then it checks side ratios and area of all extracted contours and cleans the image inside the contour with **check\_plate** and **clean\_plate** methods. After cleaning the contour image with **clean\_plate** method, it finds all characters on the plate with **find\_characters\_on\_plate** method.  
**find\_characters\_on\_plate** method uses **segment\_chars** function to find the characters. It finds characters by computing the convex hull of the contours of a thresholded value image and drawing it on the characters to reveal them.  
**Code:**Make another class to initialize Neural Network to predict the characters on the extracted license plate.

|  |
| --- |
| class OCR:        def \_\_init\_\_(self):            self.model\_file = "./model / binary\_128\_0.50\_ver3.pb"          self.label\_file = "./model / binary\_128\_0.50\_labels\_ver2.txt"          self.label = self.load\_label(self.label\_file)          self.graph = self.load\_graph(self.model\_file)          self.sess = tf.Session(graph = self.graph)        def load\_graph(self, modelFile):            graph = tf.Graph()          graph\_def = tf.GraphDef()            with open(modelFile, "rb") as f:              graph\_def.ParseFromString(f.read())            with graph.as\_default():              tf.import\_graph\_def(graph\_def)            return graph        def load\_label(self, labelFile):          label = []          proto\_as\_ascii\_lines = tf.gfile.GFile(labelFile).readlines()            for l in proto\_as\_ascii\_lines:              label.append(l.rstrip())            return label        def convert\_tensor(self, image, imageSizeOuput):          """          takes an image and tranform it in tensor          """          image = cv2.resize(image,                             dsize =(imageSizeOuput,                                    imageSizeOuput),                             interpolation = cv2.INTER\_CUBIC)            np\_image\_data = np.asarray(image)          np\_image\_data = cv2.normalize(np\_image\_data.astype('float'),                                        None, -0.5, .5,                                        cv2.NORM\_MINMAX)            np\_final = np.expand\_dims(np\_image\_data, axis = 0)            return np\_final        def label\_image(self, tensor):            input\_name = "import / input"          output\_name = "import / final\_result"            input\_operation = self.graph.get\_operation\_by\_name(input\_name)          output\_operation = self.graph.get\_operation\_by\_name(output\_name)            results = self.sess.run(output\_operation.outputs[0],                                  {input\_operation.outputs[0]: tensor})          results = np.squeeze(results)          labels = self.label          top = results.argsort()[-1:][::-1]            return labels[top[0]]        def label\_image\_list(self, listImages, imageSizeOuput):          plate = ""            for img in listImages:                if cv2.waitKey(25) & 0xFF == ord('q'):                  break              plate = plate + self.label\_image(self.convert\_tensor(img, imageSizeOuput))            return plate, len(plate) |

It loads the pretrained OCR model and its label file in **load\_graph** and **load\_label** functions. **label\_image\_list** method transforms the image to tensor with **convert\_tensor** method and then predicts the label of tensor with **label\_image\_list** function and returns the license number.  
**Code:**Create a main function to perform the whole task in a sequence.

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":        findPlate = PlateFinder()      model = OCR()        cap = cv2.VideoCapture('test\_videos / video.MOV')        while (cap.isOpened()):          ret, img = cap.read()            if ret == True:              cv2.imshow('original video', img)                if cv2.waitKey(25) & 0xFF == ord('q'):                  break                possible\_plates = findPlate.find\_possible\_plates(img)                if possible\_plates is not None:                    for i, p in enumerate(possible\_plates):                      chars\_on\_plate = findPlate.char\_on\_plate[i]                      recognized\_plate, \_ = model.label\_image\_list(                                 chars\_on\_plate, imageSizeOuput = 128)                        print(recognized\_plate)                      cv2.imshow('plate', p)                        if cv2.waitKey(25) & 0xFF == ord('q'):                          break          else:              break        cap.release()      cv2.destroyAllWindows() |