Vehicle plate recognition

Group member: Huachao Lin

Abstract

Vehicle plate recognition is a useful tool at traffic tolls and auto enforcement of traffic violations. The goal of this project is to recognize the plate of each photo in a precise way. Two datasets[1,2] of cars' images with license plates captured are used to perform plate recognition in this project. Both datasets are images of plates in the USA. The plate's location is given by x,y coordinates, length and width of the plate in each corresponding txt file. The photos are taken from different angles and at different distances. The resolution of plates is different from each other as a consequence. The project mainly consists of two parts: plate segmentation and plate letter recognition. Contour algorithm, together with some constraints, is used for segmentation. The letters in plates are formed by numbers '0-9' and letters 'A-Z', which is 36 categories in total. Random forest, neural network and etc methods are used to build single letter recognition model with a dataset[3] of 36576 28*28 standard images of 36 categories. The model with best accuracy is used to recognize each letter in a plate. Plate letters are combined and compared with the true label to evaluate the plate recognition accuracy.

2 Dataset and Methodology

The vehicle plate dataset A[1] consists of 751 photos of cars with license plates. The vehicle plate dataset B[2] consists of 222 photos of cars with license plates. The plate's location is given by x,y coordinates, length and width of the plate in each corresponding txt file. With this information, each plate can be cut out as a matrix form. The resolution of plates, or the size of matrix, is different from each other as a consequence. The resolution and angle of plates might be different from each other such as shown in Figure 1. Some images in dataset B were taken far away from the vehicle and the resolution for the plate is very low, 10 by 40 for a whole plate, for example. This kind of plate is hard to recognize, even by humans.

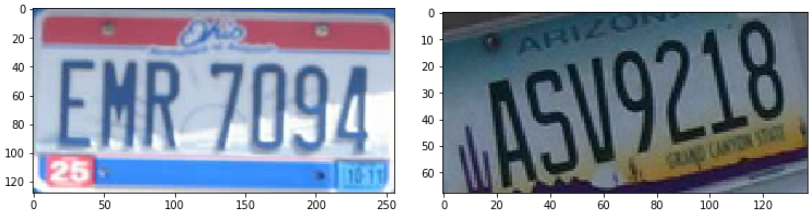


Figure 1. Plate images at different distances and angles.

After getting the information for each plate in each image. The plate images are then processed to a binary form. The processed images then are segmented into separate characters

and then recognized by each character. The details of each step and algorithms applied for each step is shown as below.

2.1 Plate Image Pre-processing

2.1.1 Convert to grayscale image

Once we have cropped the (color) plate image from the whole image, the next step is to to image thresholding to convert the image to a binary image with only white and black pixels. Before that, the color plate image (3 channels) has to be converted to a gray-scale image with 1 channel. This can be done with the OpenCV tool: cv2.cvtColor(). See Figure 2.

```
gray = cv2.cvtColor(img_plate, cv2.COLOR_BGR2GRAY)
```

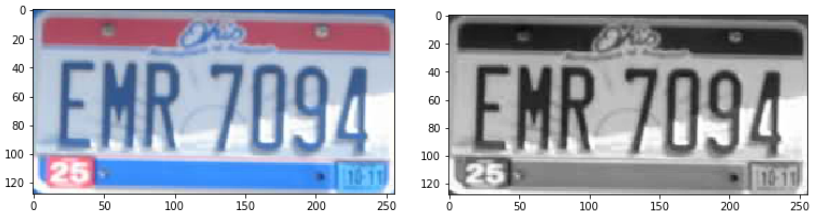


Figure 2. Convert color image (left) to gray-scale image (right)

2.1.2 Image binarization: thresholding and color inversion

After that, image thresholding can be performed on the gray-scale image. There are several approaches available to thresh an image. In this project, I tried two different methods: (1) adaptive thresholding, and (2) OTSU's thresholding. See Figure 3. There two approaches perform very similar with each other in terms of highlighting out the digits in the license plate. However, in the segmentation and prediction part, it affects the segmentation and prediction performance depends on which thresholding method is used. Shades, dirty noise might impact the image and we want the thresholding to get off these noise and the boundary of each letter is clear and correct. After trial-and-error, I found that overall the OTSU's thresholding algorithm outperforms the Adaptive thresholding algorithm. Thus, OTSU's thresholding method is adopted in this project.

```
image_to_thresh = gray
# adaptive thresholding
binary_inv_adaptive = cv2.adaptiveThreshold(image_to_thresh, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 39, 1)
# OTSU's thresholding
ret, binary_inv_otsu = cv2.threshold(image_to_thresh, 0, 255,
cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
```

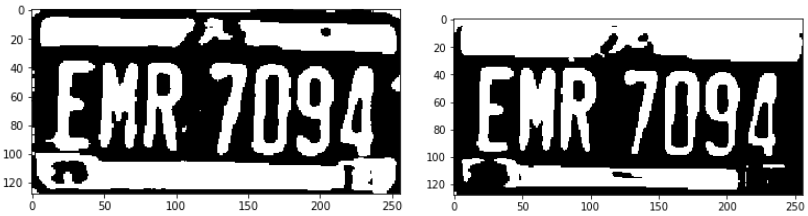


Figure 3. Adaptive thresholding (left) and OTSU's thresholding (right)

Note: in the above image thresholding, the binary image (after thresholding) is inverted. The reason for the color inversion is because the training alphabet dataset we use in the model training part has white pixel digits on a black pixel background. Since OpenCV allows color inversion with thresholding, these two steps are combined in the same line of code.

The above procedure can handle most common cases for thresholding and inversion. However, there are some special cases that the above procedure is not adequate and may fail.

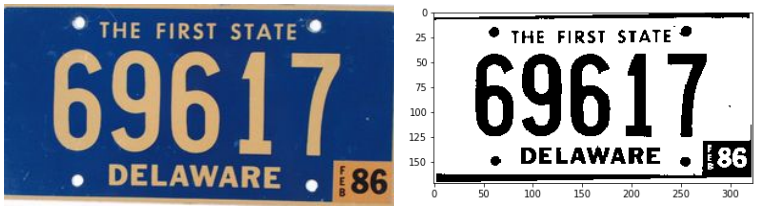


Figure 4. A DE license plate (left); After thresholding and inversion (right).

Figure 4 is a typical DE state license plate. It is not like many other states of which the license plates have a light color background. This license plate features a dark color background. After applying the above thresholding and inversion process, we will end up with the binary image in the right of Figure 4. As we discussed before, this type of binary image is not suitable for this project. Therefore, it needs to be inverted again. In order to determine if a license plate is a "DE" style, I develop a technique to automatically accomplish that. In the above binary image, the majority of the pixels are white pixels. And in a "regular" binary image (like the ones in Figure 3), the majority of the pixels are black. So we could count the total number of white and black pixels in a binary image, and compare them. If white > black, then we should invert the image once more.

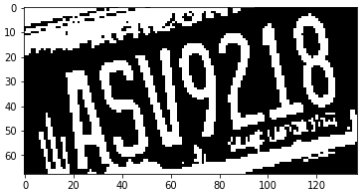
```
# in case the binary_inv_otsu image is in white background, we inverse it again
total_pixels = binary_inv_otsu.shape[0] * binary_inv_otsu.shape[1]
total_white = cv2.countNonZero(binary_inv_otsu)
total_black = total_pixels - total_white
if(total_white > total_black):
    binary_inv_otsu = cv2.bitwise_not(binary_inv_otsu)
```

2.1.3 Image rotation

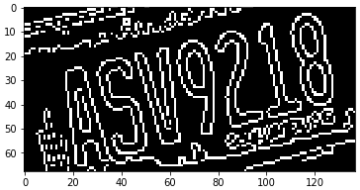
As we mentioned before, there might be some license plate images that are in weird orientation. Digits/letters on the license plate might not be in the up-right position. For those images, rotation is needed to bring them to the correct orientation.

In order to do the rotation, the HoughLine transformation[5] technique is used. The ideas of HoughLine Transformation is to find a line that connects multiple feature points in an image. The image used to find HoughLines should be carefully selected, as it has a huge impact on the result of HoughLine Transformation. Worst case is that too many HoughLines are detected on an image, and none of them are catching the correct information of the rotation of the image. After some tests, I found that using Canny[6] images after OTSU's thresholding and inversion can produce the best HoughLine Transformation result. We can see in Figure 5(f), the original license plate image has been rotated to the right position in which the digits in the image are all in the up-right position.

(a) Plate image

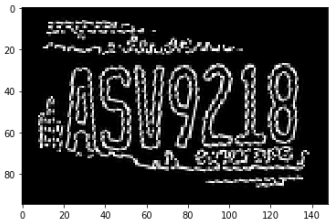


(b) OTSU's thresholding and inversion

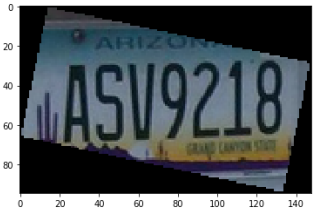


(c) Canny on OTSU's thresholding and inversion

(d) HoughLine on plate image



(e) Canny image after rotation



(f) Plate image after rotation

Figure 5. Steps for image rotation

```
edges = cv2.Canny(binary_inv_otsu, 50, 200, apertureSize = 3)
lines = cv2.HoughLines(edges, 1, np.pi/180, 2)

for rho,theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
```

```
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    cv2.line(draw_hough_line,(x1,y1),(x2,y2),(0,0,255),2)

if(180*lines[0][0][1]/3.1415926 < 20):
    img_plate_rotated = ndimage.rotate(img_plate, 180*lines[0][0][1]/3.1415926)
elif(180*lines[0][0][1]/3.1415926 > 160):
    img_plate_rotated = ndimage.rotate(img_plate, 180-180*lines[0][0][1]/3.1415926)
else:
    img_plate_rotated = ndimage.rotate(img_plate, 180*lines[0][0][1]/3.1415926-90)
```

2.2 Plate segmentation

2.2.1 Find contours

Now the plate image is in the right position, we can move on to the next step: segmenting the digits in the rotated image. In this step, I adopted the cv2.findContour() method to find all the contours in the binary image. As we can see in Figure 6, we successfully located the contours of all the digits in the image. However, there are also many other contours identified in this image (see the small irregular shape in Figure 6). These contours will be removed in the next contour cleaning step. The ideal result should be that only the contours of the license plate digits are kept.

```
contours = measure.find_contours(image, 0.5)
```

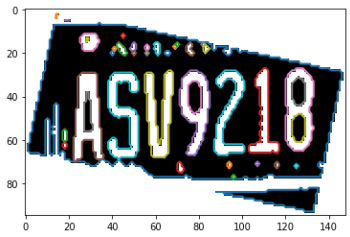


Figure 6. Contours in the binarized image.

2.2.2 Filter contours by different criteria

Contours are basically points on the boundary of some connected pixels in an image. The points have the leftmost point, the rightmost point, the topmost point and the bottommost point. These points together can form a bounding box of the contour. Therefore, we could easily obtain the width and height of a contour.

Not every contour is a letter/number. In fact, characters in a license plate have very unique characteristics. For example:

(1) Y-position: in most of the cases, digits should be located approximately near the horizontal centerline of a license plate in terms of y-position. Any contours that are far away from the horizontal centerline should not be considered as a digit in the license plate.
(2) Contour areas: Also, the contour of a digit in a license plate should neither be too small nor too large. Any contour that is smaller than a lower bound or larger than an upper bound should also not be considered as a digit in the license plate.
(3) The ratio of width and height: All digits (0-9, A-Z) should have a reasonable width/height ratio. Anything, for example a line, that is very narrow in width would have an extremely small width/height ratio. By applying this filter, we can confidently filter out such contours.
(4) The height of the contour: All digits (0-9, A-Z) should have a reasonable height. From my testing experience, any digits should have a height greater than (>) at least 0.3 * the height of the license plate image. Anything smaller than that threshold should not be considered as a digit in the license plate.

```
    with_boxes  = np.copy(img_plate_rotated)

    digits_cropped = []
    last_contour_bbox = [0, 0, 0, 0]
    for n, contour in enumerate(sorted_ctrs):
        Ymin = int(np.min(contour[:,0]))
        Ymax = int(np.max(contour[:,0]))
        Xmin = int(np.min(contour[:,1]))
        Xmax = int(np.max(contour[:,1]))
           w_ = Xmax - Xmin
        h_ = Ymax - Ymin
        area_ = w_ * h_
        Y_center = (Ymin + Ymax) / 2
```

```
# we set three.four evaluation criteria to filter out contours that are not digits
        # 1: min_area < area size < max_area
        # 2: width and height ratio > w_h_ratio
        # 3: bbox_center_lo < center point < bbox_center_hi
        # 4: the height of the digit has to be greater than h_ratio * h_image

    min_area = area * low
    max_area = area * up
    w_h_ratio = 0.1
    bbox_center_lo = 0.2 * h
    bbox_center_hi = 0.8 * h
    h_ratio = 0.3
    if(area_ > min_area and area_ < max_area and
           w_/h_ > w_h_ratio and h_/w_ > w_h_ratio and
           Y_center > bbox_center_lo and Y_center < bbox_center_hi and h_ > h_ratio*h):
```

2.2.3 Sort contour to the correct order

The contours detected from the license plate image are not necessary in the correct order as the digits in the image. Before we move on to the recognition step, we have to put them in the right order. Since we know the bounding box of a contour, we can sort the contours by Xmin, which is the x coordinates of the top-left corner of the bounding box.

```
sorted_ctrs = sorted(contours, key=lambda contour: np.min(contour[:,1]))
```

2.2.4 Some extra filtering

Sometimes, a digit can get two or more contours. For example the “0 (zero)” and “O (O as in Okay)” digits. See Figure 7. The outer boundary of the digit will get an outer contour, while the inner boundary of the digit will also get a contour. This is due to the characteristic of such characters. So we have to remove such inner contours to avoid duplicate detections. The technique I used is very simple by checking if a bounding box is completed surrounded by the last bounding box in the sorted contours. If so, we remove this bounding box. Otherwise, we keep it.

```
if(Xmin > last_contour_bbox[0] and
   Xmax < last_contour_bbox[1] and
   Ymin > last_contour_bbox[2] and
   Ymax < last_contour_bbox[3]):
```

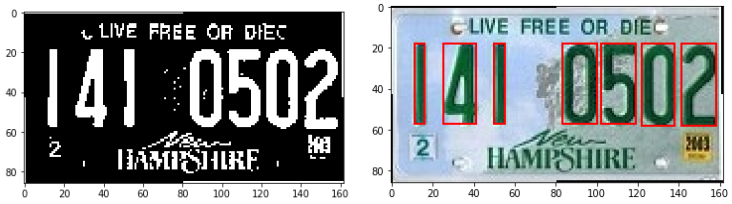


Figure 7. Some digits will get multiple contours (the number 0 gets two contours in the example).

2.2.5 Padding and image resizing

Now we have successfully identified the bounding boxes for digits, and removed those undesired bounding boxes, it is time to recognize what digit is in each bounding box. We can crop the bounding box from the plate image, and perform prediction based upon the cropped image. Notice that the shapes of the bounding boxes can vary significantly. See Figure 8. Before we feed the cropped image to a prediction model, we need to first resize the dropped images to the same size as the input size of the prediction model.

Figure 8. The size of the bounding box can vary.

In order to do resizing, squaring the digit image is needed. Since bounding boxes are usually in rectangular shape instead of square. We need to make it square by putting zeros around the (binary) digit image. Then we do padding the digit image. Although the digit image is squared, the digit in the squared image is very close to the boundary. This would deteriorate the prediction accuracy as the digits in the training dataset all have a certain amount of black margin around the digits. See Figure 9.

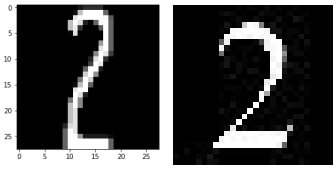


Figure 9. Digit image after squaring (left); image from training dataset (right).

So we should add such black margins to the image by adding a padding to the image. Then we resize the image. Every padded digit image is resized to a 28x28 square image with cv2.resize function. The image after squaring, padding and resizing will be as similar to below.

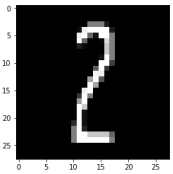


Figure 10. The digit image after squaring, padding and resizing.

2.3 Character recognition

Recognition models need to be built to classify a 28*28 images into 36 categories. A dataset[3] of 36576 28*28 standard images of 36 categories are used to train and validate our model. The number of samples for each class is 1016. The model is not limited to recognizing the character in our plate image but can be applied to any other plate dataset formed by numbers and letters. But the type of noise might differ from dataset to dataset. The type of noise in the plate images is not exactly the same as the one we built the model on. This makes our classification model more generalized but might lose some accuracy in plate recognition. The model with best accuracy is used to recognize each letter in a plate and the letters in a plate are compared with the true label to evaluate the plate recognition accuracy.

Random forest, neural network, and etc methods are used to build the single letter recognition model. One hot encoder is used to encode the y labels from ‘0-9A-Z’. For

Convolutional Neural Network, X input as a 28*28 matrix. For others, the X need to be reshape to 784 instead a 28*28 matrix. Train test data is splitted into an 8:2 ratio. Random forest classifiers are applied with several n_estimators turned in the model. T-Distributed Stochastic Neighbor Embedding (TSNE) unsupervised learning is used to transform the X into 2 dimensions input. Together with random forest, one classification model is trained and accuracy is calculated. Convolutional Neural Network(CNN) is an advanced model often with high accuracy. This project also uses this method to build a classification model.

CNN models are well known as a great model for detecting shapes, objects, numbers in an image. It is fast and produces relatively high accuracy in prediction compared to other machine learning techniques. Our task in this project is very similar to the popular MNIST digit recognition challenge. In this project, we need to recognize 36 classes of characters (0-9, A-Z), while the MNIST challenge only needs to recognize 10 classes of characters (0-9). The methodology should be similar.

In this project, I choose to build a multi-layer CNN model consisting of: convolutional layer, max_pooling layer and drop_out layer. The structure of this CNN model is inspired by Achraf KHAZRI[4]. Model summary is as following:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_64 (Conv2D)           (None, 28, 28, 32)        320
_________________________________________________________________
conv2d_65 (Conv2D)           (None, 26, 26, 32)        9248
_________________________________________________________________
max_pooling2d_27 (MaxPooling (None, 13, 13, 32)        0
_________________________________________________________________
dropout_36 (Dropout)         (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_66 (Conv2D)           (None, 13, 13, 64)        18496
_________________________________________________________________
conv2d_67 (Conv2D)           (None, 11, 11, 64)        36928
_________________________________________________________________
max_pooling2d_28 (MaxPooling (None, 5, 5, 64)          0
_________________________________________________________________
dropout_37 (Dropout)         (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_68 (Conv2D)           (None, 5, 5, 64)          36928
_________________________________________________________________
conv2d_69 (Conv2D)           (None, 3, 3, 64)          36928
_________________________________________________________________
max_pooling2d_29 (MaxPooling (None, 1, 1, 64)          0
_________________________________________________________________
dropout_38 (Dropout)         (None, 1, 1, 64)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_18 (Dense)             (None, 512)               33280
_________________________________________________________________
```

```
dropout_39 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_19 (Dense)             (None, 36)                18468
=================================================================
Total params: 190,596
Trainable params: 190,596
Non-trainable params: 0
```

The CNN model is built with the keras library using a tensorflow backend. To speed up the training process, I opt for GPU training. For comparison purposes, one epoch (batch size=64) takes 6-7s to train on my GPU (GTX 1050 Ti with Max-Q), while it takes 35-38s to train on my CPU (intel i7-8850H). Deep learning framework configuration in this project is:

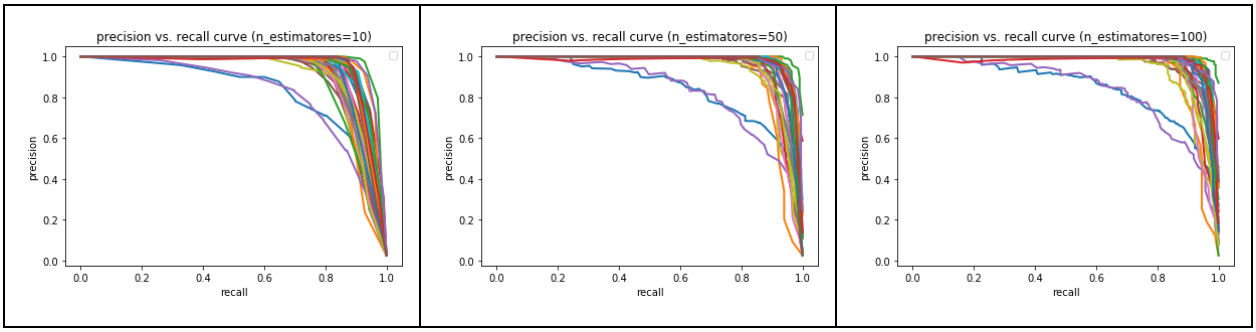
```
Tensorflow: 2.2.0
CUDA: 10.2.89
CuDNN: 7.6
```

3 Results

3.1 Character Recognition Model Result

Random forest, TNSE transformation with random forest and convolution neural network methods are applied to build the single character recognition model.

The accuracy rate, PR curve and ROC curve are compared in random forest with the different number of trees in the forest. 100 trees shows a higher accuracy rate than 10 and/or 50 trees, achieving a 75.5% accuracy rate. The details are shown in Figure 7. The accuracy did not improve a lot from 10 trees to 100 trees in forest, only from 71.8% to 75.5%. The PR curve and ROC curve shows precision/recall and tp/fp relation for each class as shown in Figure 11. Generally speaking, the closer a PR is to the upper right corner and the closer a ROC is to the upper left corner, the better the test is. It is obvious that the PR of two classes is not so close to the upper right corner and this might be the reason for the accuracy of random forest only reaching 75.5%.



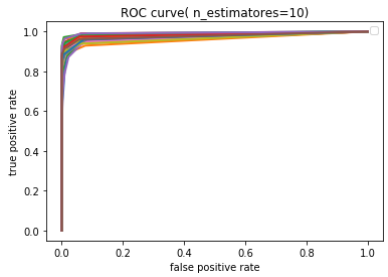
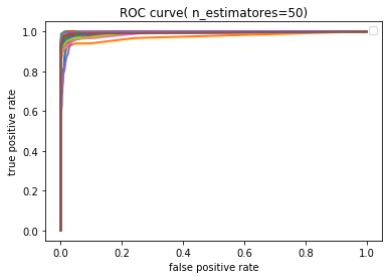
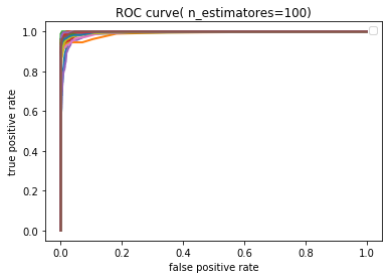
		
Accuracy rate: 71.8%	Accuracy rate: 75.2%	Accuracy rate: 75.5%

Figure 11. PR, ROC curve and accuracy for random forest with different trees in forest

From Figure 12 we can see how TNSE reduces the dimension while separating the clusters in the meantime. After transforming the data by TNSE unsupervised learning method, random forest is used to do the supervised learning step with 36 categories. The results summary is shown in Figure 13.

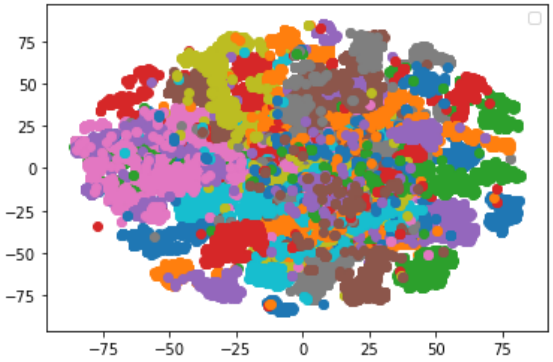


Figure 12. Clusters of 36 classes after TNSE applied

The accuracy rate of this method is 98.1% with perfect PR and ROC curve. But the problem with this method is that the whole dataset needs to be embedded at first and it is hard to transform the new dataset from external.

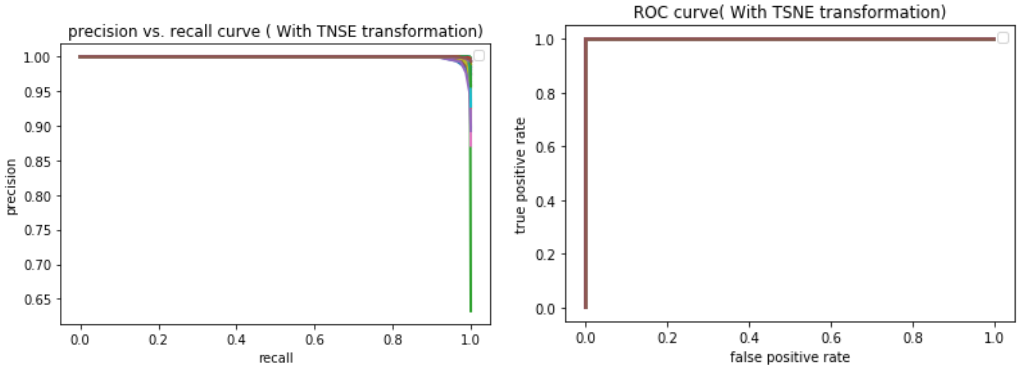


Figure 13. PR and ROC Curve of random forest with TSNE

A convolution neural network model is built to train this dataset. Optimizer is ‘Adam’ and loss is ‘categorical cross entropy’. Several activation functions, depth and width networks are trained and tested and highest accuracy one reaches an accuracy rate of 99.3%. The detail of this model is shown below. PR and ROC curve is shown in Figure 10 and the accuracy in the test dataset for each class is shown in Table 1. The ROC curve seems a perfect right angle. However, in the PR curve, there are obviously two classes not so close to the upper right corner. From the accuracy for each class, we can observe that ‘0’ and ‘O’ achiches at a low accuracy rate, 0.880 and 0.694 respectively. It is not hard to understand that 0 is similar to O and hard to recognize.

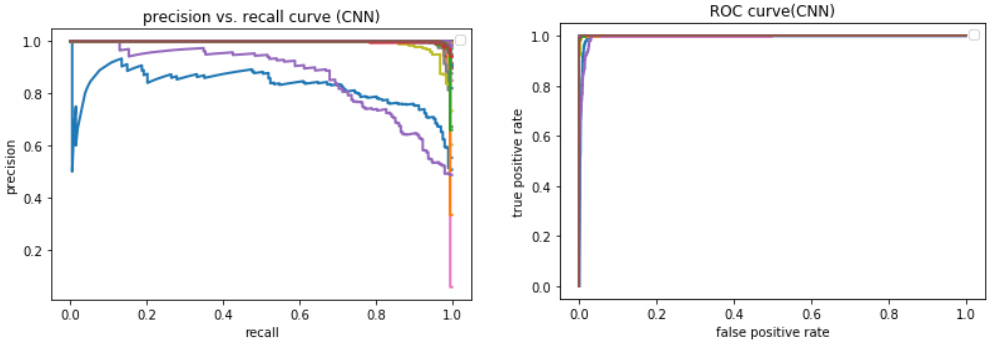


Figure 14. PR and ROC Curve of Convolution neural network model

Table 1. Accuracy rate of one to other category

Label	0	1	2	3	4	5	6	7	8	9
Accuracy	0.880	0.960	1.00	1.00	0.995	0.995	0.991	0.995	0.991	1.00
Label	A	B	C	D	E	F	G	H	I	J
Accuracy	0.990	0.984	0.990	0.995	0.995	0.980	0.972	0.958	0.974	0.995
Label	K	L	M	N	O	P	Q	R	S	T
Accuracy	0.991	0.984	0.995	0.966	0.694	0.991	0.981	0.995	0.990	0.995
Label	U	V	W	X	Y	Z				
Accuracy	0.995	1.00	0.990	0.995	0.995	0.986				

3.2 Plate Recognition Evaluation

Notice plate recognition is hard. Once one character in a plate is wrong, the whole plate recognition is wrong. In order to evaluate our model in plate recognition, we evaluate two sets of plates. Image set A contains 751 images and Image set B contains 222 images. They are both plates of vehicles in the USA with different color, plate length and other characteristics. Evaluating results for these two datasets is as shown below in Table 2.

Table 2. Accuracy of recognition

Image set	A	B
Number of images	751	222
corrects	279	22
Accuracy	37.2%	9.9%

Image set A contains whole license plate images, while image set B contains images of a vehicle with a license plate.

Figure 15. Sample images from image set A

Figure 16. Sample images from image set B

We can see from Figure 15 and Figure 16 that the resolution of the license plate in the image is quite different. The plate image quality is significantly higher in image set A than in image set B. Therefore, the detection accuracy for image set A is also significantly higher than that of image set B (37.2% vs. 9.9%). In fact, the accuracy for plate recognition is low. This is understandable due to the various license plate styles throughout the United States. Different states have different license plate styles, and usually the styles are quite different with others. This complicates the segmentation and detection process. It is very difficult to develop a simple detection and prediction scheme to work well for every state's license plate. During the work of this project, I found that through fine-tunings of different parameters for thresholding/findContour/resizing/etc., it will improve the detection accuracy by a certain amount. However, due to the limited time and resources available, there is not enough time for me to fine-tune every parameter to make the model perfect. I will keep the model as is for the course project, and will continue to improve it in the future.

4 Conclusion

The classification model for '0-9' 'A-Z' reaches a high accuracy. Random forest with TNSE reaches a 98% accuracy rate and CNN reaches a 99% accuracy rate. The classification between '0' and 'O' is hard and gets false recognition frequently.

The pipeline and recognition model perform smoothly. Some plates are successfully segmented and recognized. However, low accuracy is reached on these two datasets, some coming from the noise coming from shadow, dirty or different angle and distance. The main problem, however, is the complexity of vehicle plates in the USA. Some special patterns such as grasses, trees might connect with letters/numbers and make segmentation especially hard by contour algorithm. There is also some error caused by some bad quality plates, such as images for vehicles from far away and the plate is very small.

5 Future work

The segmentation and detection processes are complicated by the complexity of plates in the USA. It is very difficult to develop a simple detection and prediction scheme to work well for every state's license plate. Through fine-tunings of different parameters for thresholding/findContour/resizing/etc., it will improve the detection accuracy by a certain amount. However, due to the limited time and resources available, there is not enough time for me to fine-tune every parameter to make the model perfect. I will keep the model as is for the course project, and will continue to improve it in the future.

Also, the CNN model used in this project is not a very sophisticated model. It could be improved by better designing the structures of the CNN model. Furthermore, we could also try out different optimizers in the training process. The training epochs should also be carefully picked to achieve optimal detection results, and to avoid over training.

6 Reference

[1] Image set A: https://github.com/openalpr/benchmarks/tree/master/seg_and_ocr/usimages
[2] Image set B: https://github.com/openalpr/benchmarks/tree/master/endtoend/us
[3]https://github.com/GuiltyNeuron/ANPR/blob/master/Licence_plate_recognition/USA_plates/dataset.zip
[4] A CNN model:
https://github.com/GuiltyNeuron/ANPR/blob/master/Licence_plate_recognition/USA_plates/train.py
[5] Hough line transform:
https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html
[6] Canny edge detection:
https://docs.opencv.org/2.4/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html