Vehicle Detection and License Plate Recognition using Deep Learning

ENSC424 Final Project
Professor: Jie Liang

Group11
Arlene Fu, 301256171
Ricky Chen, 301242896
Toky Saleh, 301160773
Karamveer Dhillon, 301209928

Table of Contents

Contents

Table of figure

1. Introduction

1.1 Background

Deep learning has been a huge topic in machine learning research closer to Artificial Intelligence. A few decades ago, computer scientists developed a number of algorithms that trained computers to distinguish multiple instances of the same object.

With the growing number of drivers, and an increase of vehicles on the road come problems associated with traffic. Some of these problems, such as accurate bridge and highway tolling, parking lot management, and speed prevention, can now be solved using machine learning. This project will explore the use of deep learning for the purpose of vehicle tracking and license plate recognition.

This project gave us the basic understanding of the modern neural network and how it works with applications in computer vision. By using the building blocks of neural networks, we were able to improve the accuracy of a model with its pre-trained model. We used our understanding of the pre-defined building model of the neutral network to compare the model accuracy using TensorFlow and Keras framework.

1.2 Literature Review

There are many published papers that discuss topics about vehicle detection. We will focus on one article named 'Image-based vehicle analysis using deep neural network: a systematic study' [1]. In this article, they use a DNN (YOLO) model to achieve vehicle detection. The article also covers training for vehicle classification on normal images and dark images. We are able to infer the success of our chosen project through the study of this article.

2. Project Description

Our project is divided into two major components: vehicle detection and license plate character recognition.

Phase I is detecting whether an input contains vehicles or not. If the algorithm predicts that the input contains vehicles, then we need to define a method which will precisely locate and crop the vehicle from the original data. Starting with the output from Phase I as the input, Phase II should search for the vehicle's license plate and have the ability to recognize the number on the car plate.

Through investigating, we found that vehicle detection is already quite developed and there are numerous existing methods online; however, we still decided to explore deep learning through this project. We attempt to improve the accuracy with some pre-trained models.

What's more, in consideration of time and resource limitation, (we neither have enough time nor access to any GPU machine), we decide to challenge image based detection rather than video based.

2.1 Vehicle Detection

2.1.1 Initial Plan

We first decided to build our own CNN model while using ImageNet as our starting point. We also considered applying some data augmentation on our dataset as a feasible method to improve prediction accuracy. Based on this idea, we did some research on currently popular solutions for vehicle detection.

2.1.2 YOLO Model

YOLO caught our attention during our online investigation, as it was the most popular method of real-time object detection. Unlike prior detection systems, which apply the model to a target at multiple locations with different scales, YOLO only applies a single neural network to the whole target image. According to the YOLO official website, the image is divided into regions with different weighted predicts bounding boxes. Each region has its own probability. Compared to previous work for object detection like R-CNN, which requires recursion and needs a lot of time and memory, YOLO only looks once at the whole image. Therefore, it is able to digest more global context in the image which makes it work extremely fast.

Since 'car' is a known class for YOLO, we decided to use this method. We found some valuable code written by Junsheng Fu from GitHub [1], which gave us insight into the use of YOLO weights to detect objects and how to implement visualization of the detected results. The available code uses TensorFlow implementation with pretrained YOLO_small weight.
Based on the code we found, we first made a single non-greyscale image as input data, and tried to test whether the code performed detection well or not. Every image that passes into the program is resized to 448x448, in order to match the model. In the meanwhile, we found that OpenCV gives wrong color to colored images when loading. This is because OpenCV uses BGR as its default color order for images while matplotlib uses RGB. When displaying an image loaded by OpenCV in matplotlib, the colors will be incorrect. This problem requires the use of OpenCV to explicitly convert data back to RGB:

RGB_img=cv2.cvtColor(BGR_img,cv2.COLOR_BGR2RGB)

We also implemented the visualization part to mark the detected vehicle on the input image and cropped it for future use. Our implementation is able to crop multiple detected cars as well (see Figure 1).

According to the source code, the author defines his own layers rather than using Keras layers and sets up a convolutional neural network model. We wanted to apply YOLO_tiny weight from the YOLO official website to the model schema in Figure 2 by using a similar method, but the result was unsuccessful. The reason for this failure is that the weight from YOLO website is designed for a specific model which is built in a .cfg file. In the .cfg file, the model is not sequential. After some investigation, we cannot find any Keras parallel model. For the purpose of using pre-trained weight, we needed to convert the .cfg format model into the model suitable for Keras through darkflow. This sample code[3] helped us to do this job. Due to the time limitation of our project, we determined not to use this method.

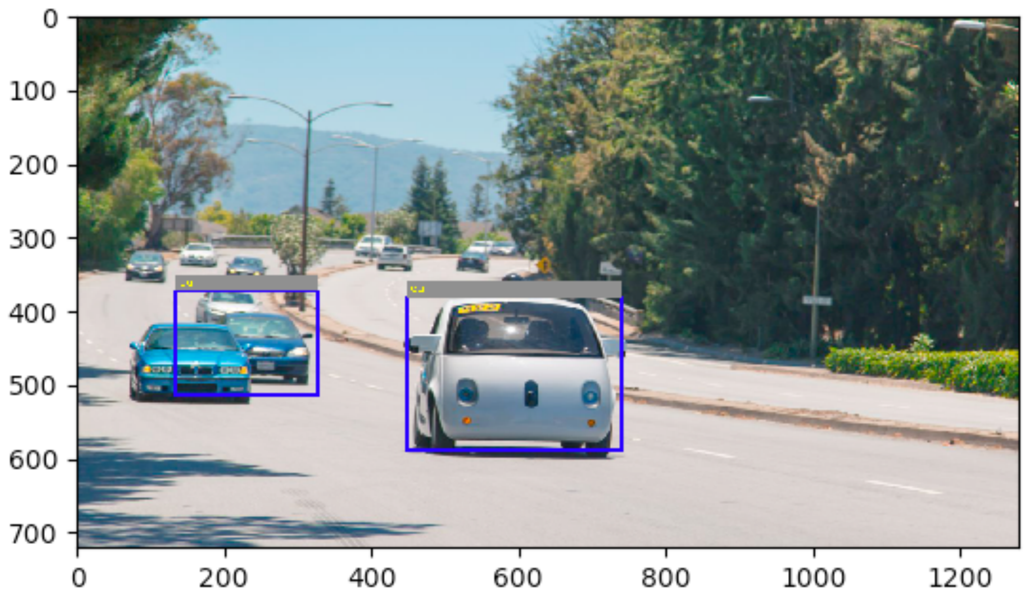
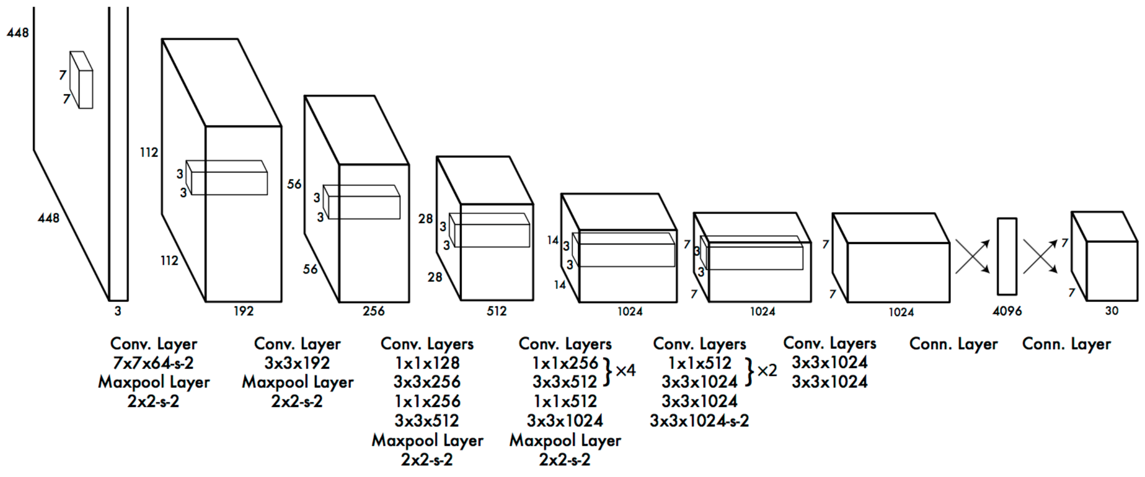


Figure 1. Result of YOLO detection



Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 2. Tiny YOLO model [2]

Instead, we passed in 8000 test images through a loop. The images were all from a 'cars dataset' published by Stanford University [4]. This is a large-scale, fine-grained dataset of cars. The average image size of this dataset is 980 x 728, and the largest image resolution is 3280 x 2240. The reason for choosing this dataset over some other small dataset is that we needed to crop the detected vehicle and pass the output to process the license plate for character recognition. If the resolution of initial input is too low, the whole program will lose accuracy. So, there is a tradeoff between speed and performance. Large images required more training/testing time and a more powerful device, but it gave a higher chance for us to detect a car and the car plate.

But a problem arose while using the high-resolution images as a test set; the memory was not large enough to accommodate them. So, after looping the whole program 32 times (in order to process 32 images), the system crashed and read the 33rd image as a grayscale image.

2.1.3 YOLO Custom Dataset

Since the modification of neural network structure is limited by time and knowledge, we decided to adopt the common YOLO network structure, but we trained it from scratch with our custom dataset. We were interested in the extent of improvements via training with the relatively small number of data.

Some specifications need to be clarified before proceeding into this project. First, YOLO program is implemented on the Linux platform, which means that modifications and debugging will be easier in Linux instead of Windows. Second, several applications are required to execute the files, including Python 2.7, OpenCV and the Darknet Toolkit. Notice that neither a virtual environment nor a specific backend is required since the YOLO configuration file is already implemented. Third, due to conflicts of the Nvidia graphic driver on a dual booting system, the CUDA toolkit cannot be utilized here, which in other words, means we cannot use a GPU to perform training. Lastly, in order to save time on data processing, we will focus on image dataset rather than real-time detection.

2.1.4 Data Preprocessing

The first challenge of the project was preprocessing our custom datasets. The dataset was obtained from the Stanford website [25]. This dataset contained both a training set and a validation set, and each set contained more than 8000 car images. YOLO detection features the way of bounding boxes as shown in Figure 1. So, for car detection, we needed to manipulate our training sets and generate the information of each bounding boxes.

We used the BBox-labeling tool to manually draw the bounding box [7][26]. This tool was able to record the appearances of cars in each image, and the absolute coordinates of the bounding boxes. Unavoidably, the accuracy of training depended on our drawing in this case. In addition, as mentioned in its documentation, this tool only recognized images with the suffix of ".JPEG". Therefore, non-JPG type files needed to be converted first, and file names were required to be reorganized by some naming software like Bulk Rename in Ubuntu.

However, YOLO program recognized the coordinates in a different way from the labeling tool. YOLO expected XY coordinates of both the object center and width, while the labeling tool generated XY coordinates of four vertices [7]. We then used the script called "convert.py" from Darknet to complete this conversion [9][11].

2.1.4 YOLO Configuration

Next, we needed to implement a custom YOLO configuration file in order to train our dataset efficiently. In the Darknet program, several configuration files were already given, including the whole neural network structure setup as shown in Figure 2. Therefore, there were only a few items to be modified. In the car detection phase, there was only one class which is car itself. The object data was referred to by the directories of all training images and validation images. The batch number indicated the number of images processed at each training step, which was further rectified by subdivisions according to the computer performance. The number of filters was determined by the following formula:

$$\text{Filters} = (\text{Class}+5)\times 5$$

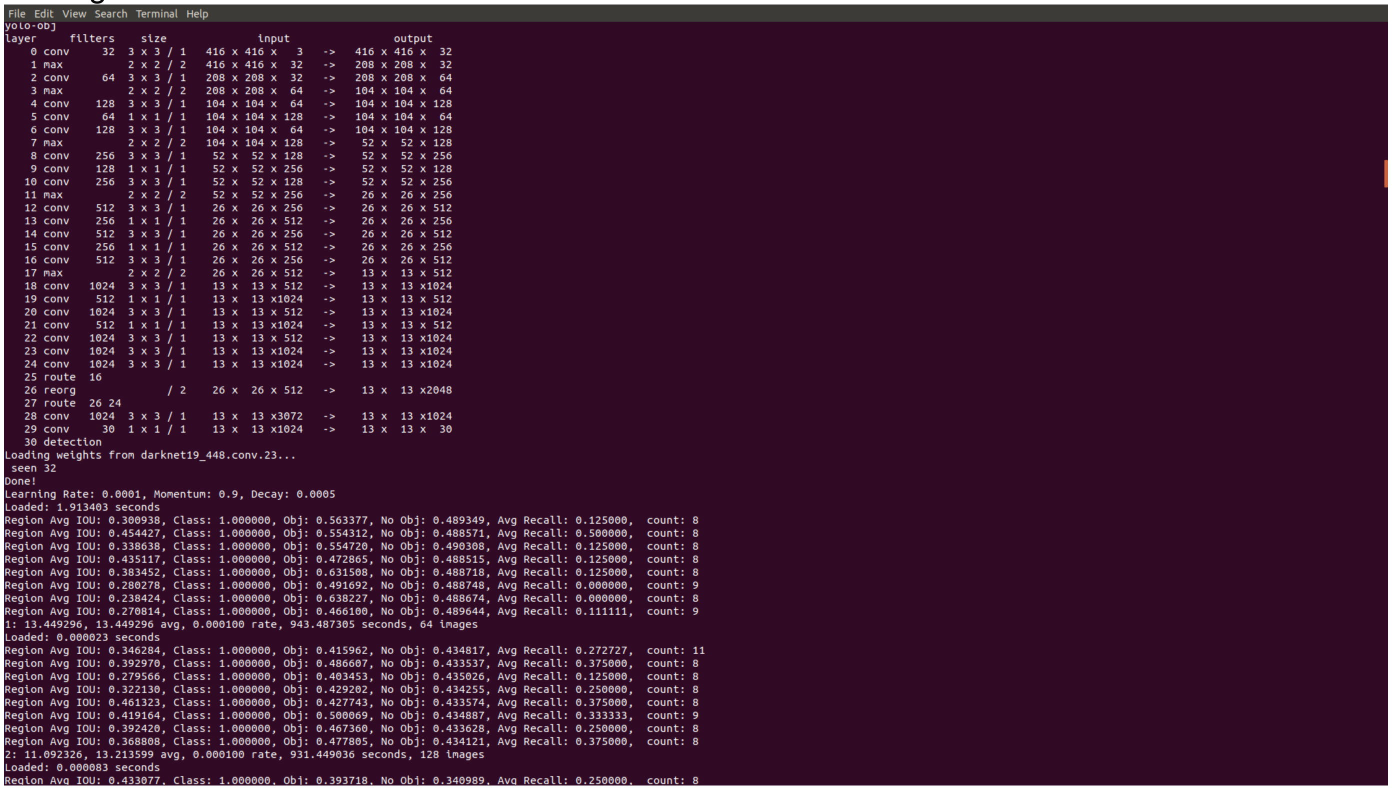
Which in our case, was equal to 30 [7]. Furthermore, we will take a pre-trained weight as a starting point. But from the testing results later on, we find this weight is basically of scratch.

2.1.5 YOLO Training

We began training by using the following command in Linux:

```
./darknet detector train <obj.data> <obj cfg file> <pre-trained weight>
```

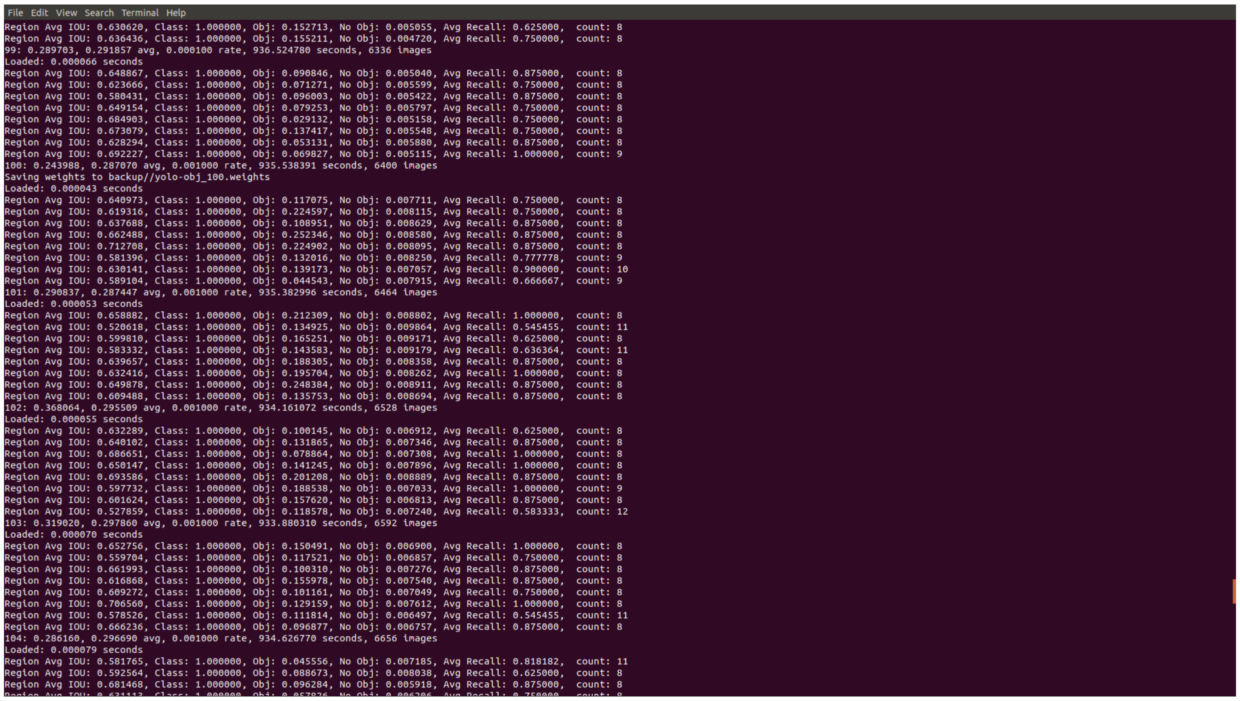
Figure 3 shows the command window after successful execution:



```
yolo-obj
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32
    1 max          2 x 2 / 2   416 x 416 x  32   ->   208 x 208 x  32
    2 conv     64  3 x 3 / 1   208 x 208 x  32   ->   208 x 208 x  64
    3 max          2 x 2 / 2   208 x 208 x  64   ->   104 x 104 x  64
    4 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128
    5 conv     64  1 x 1 / 1   104 x 104 x 128   ->   104 x 104 x  64
    6 conv    128  3 x 3 / 1   104 x 104 x  64   ->   104 x 104 x 128
    7 max          2 x 2 / 2   104 x 104 x 128   ->    52 x  52 x 128
    8 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256
    9 conv    128  1 x 1 / 1    52 x  52 x 256   ->    52 x  52 x 128
   10 conv    256  3 x 3 / 1    52 x  52 x 128   ->    52 x  52 x 256
   11 max          2 x 2 / 2    52 x  52 x 256   ->    26 x  26 x 256
   12 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512
   13 conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256
   14 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512
   15 conv    256  1 x 1 / 1    26 x  26 x 512   ->    26 x  26 x 256
   16 conv    512  3 x 3 / 1    26 x  26 x 256   ->    26 x  26 x 512
   17 max          2 x 2 / 2    26 x  26 x 512   ->    13 x  13 x 512
   18 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024
   19 conv    512  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 512
   20 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024
   21 conv    512  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 512
   22 conv   1024  3 x 3 / 1    13 x  13 x 512   ->    13 x  13 x1024
   23 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024
   24 conv   1024  3 x 3 / 1    13 x  13 x1024   ->    13 x  13 x1024
   25 route  16
   26 reorg              / 2    26 x  26 x 512   ->    13 x  13 x2048
   27 route  26 24
   28 conv   1024  3 x 3 / 1    13 x  13 x3072   ->    13 x  13 x1024
   29 conv     30  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x  30
   30 detection
Loading weights from darknet19_448.conv.23...
 seen 32
Done!
Learning Rate: 0.0001, Momentum: 0.9, Decay: 0.0005
Loaded: 1.913403 seconds
Region Avg IOU: 0.300938, Class: 1.000000, Obj: 0.563377, No Obj: 0.489349, Avg Recall: 0.125000,  count: 8
Region Avg IOU: 0.454427, Class: 1.000000, Obj: 0.554312, No Obj: 0.488571, Avg Recall: 0.500000,  count: 8
Region Avg IOU: 0.338638, Class: 1.000000, Obj: 0.554720, No Obj: 0.490308, Avg Recall: 0.125000,  count: 8
Region Avg IOU: 0.435117, Class: 1.000000, Obj: 0.472865, No Obj: 0.488515, Avg Recall: 0.125000,  count: 8
Region Avg IOU: 0.383452, Class: 1.000000, Obj: 0.631508, No Obj: 0.488718, Avg Recall: 0.125000,  count: 8
Region Avg IOU: 0.280278, Class: 1.000000, Obj: 0.491692, No Obj: 0.488748, Avg Recall: 0.000000,  count: 9
Region Avg IOU: 0.238424, Class: 1.000000, Obj: 0.638227, No Obj: 0.488674, Avg Recall: 0.000000,  count: 8
Region Avg IOU: 0.270814, Class: 1.000000, Obj: 0.466100, No Obj: 0.489644, Avg Recall: 0.111111,  count: 9
1: 13.449296, 13.449296 avg, 0.000100 rate, 943.487305 seconds, 64 images
Loaded: 0.000023 seconds
Region Avg IOU: 0.346284, Class: 1.000000, Obj: 0.415962, No Obj: 0.434817, Avg Recall: 0.272727,  count: 11
Region Avg IOU: 0.392970, Class: 1.000000, Obj: 0.486607, No Obj: 0.433537, Avg Recall: 0.375000,  count: 8
Region Avg IOU: 0.279566, Class: 1.000000, Obj: 0.403453, No Obj: 0.435026, Avg Recall: 0.125000,  count: 8
Region Avg IOU: 0.322130, Class: 1.000000, Obj: 0.429202, No Obj: 0.434255, Avg Recall: 0.250000,  count: 8
Region Avg IOU: 0.461323, Class: 1.000000, Obj: 0.427743, No Obj: 0.433574, Avg Recall: 0.375000,  count: 8
Region Avg IOU: 0.419164, Class: 1.000000, Obj: 0.500069, No Obj: 0.434887, Avg Recall: 0.333333,  count: 9
Region Avg IOU: 0.392420, Class: 1.000000, Obj: 0.467360, No Obj: 0.433628, Avg Recall: 0.250000,  count: 8
Region Avg IOU: 0.368808, Class: 1.000000, Obj: 0.477805, No Obj: 0.434121, Avg Recall: 0.375000,  count: 8
2: 11.092326, 13.213599 avg, 0.000100 rate, 931.449036 seconds, 128 images
Loaded: 0.000083 seconds
Region Avg IOU: 0.433077, Class: 1.000000, Obj: 0.393718, No Obj: 0.340989, Avg Recall: 0.250000,  count: 8
```

Figure 3. Start Training

At first, we trained with 200 testing images and more than 8000 validation images by mistake, due to the wrong implementation of wrong directories of validation dataset. Therefore, it took us 2 days to finish 100 iterations and obtain the saved weight. It's worth mentioning that Darknet only saved the weight file every 100 iterations, and according to documentation, early stopping point had been reached before the 100th iteration, as shown in Figure 4.



```
Region Avg IOU: 0.630620, Class: 1.000000, Obj: 0.152713, No Obj: 0.005055, Avg Recall: 0.625000,  count: 8
Region Avg IOU: 0.636436, Class: 1.000000, Obj: 0.155211, No Obj: 0.004720, Avg Recall: 0.750000,  count: 8
99: 0.289703, 0.291857 avg, 0.000100 rate, 936.524780 seconds, 6336 images
Loaded: 0.000066 seconds
Region Avg IOU: 0.648867, Class: 1.000000, Obj: 0.090846, No Obj: 0.005040, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.623666, Class: 1.000000, Obj: 0.071271, No Obj: 0.005599, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.580431, Class: 1.000000, Obj: 0.096003, No Obj: 0.005422, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.649154, Class: 1.000000, Obj: 0.079253, No Obj: 0.005797, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.684903, Class: 1.000000, Obj: 0.029132, No Obj: 0.005158, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.673079, Class: 1.000000, Obj: 0.137417, No Obj: 0.005548, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.628294, Class: 1.000000, Obj: 0.053131, No Obj: 0.005880, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.692227, Class: 1.000000, Obj: 0.069827, No Obj: 0.005115, Avg Recall: 1.000000,  count: 9
100: 0.243988, 0.287070 avg, 0.001000 rate, 935.538391 seconds, 6400 images
Saving weights to backup//yolo-obj_100.weights
Loaded: 0.000043 seconds
Region Avg IOU: 0.640973, Class: 1.000000, Obj: 0.117075, No Obj: 0.007711, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.619316, Class: 1.000000, Obj: 0.224597, No Obj: 0.008115, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.637688, Class: 1.000000, Obj: 0.108951, No Obj: 0.008629, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.662488, Class: 1.000000, Obj: 0.252346, No Obj: 0.008580, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.712708, Class: 1.000000, Obj: 0.224902, No Obj: 0.008095, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.581396, Class: 1.000000, Obj: 0.132016, No Obj: 0.008250, Avg Recall: 0.777778,  count: 9
Region Avg IOU: 0.630141, Class: 1.000000, Obj: 0.139173, No Obj: 0.007057, Avg Recall: 0.900000,  count: 10
Region Avg IOU: 0.589104, Class: 1.000000, Obj: 0.044543, No Obj: 0.007915, Avg Recall: 0.666667,  count: 9
101: 0.290837, 0.287447 avg, 0.001000 rate, 935.382996 seconds, 6464 images
Loaded: 0.000053 seconds
Region Avg IOU: 0.658882, Class: 1.000000, Obj: 0.212309, No Obj: 0.008802, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.520618, Class: 1.000000, Obj: 0.134925, No Obj: 0.009864, Avg Recall: 0.545455,  count: 11
Region Avg IOU: 0.599810, Class: 1.000000, Obj: 0.165251, No Obj: 0.009171, Avg Recall: 0.625000,  count: 8
Region Avg IOU: 0.583332, Class: 1.000000, Obj: 0.143583, No Obj: 0.009179, Avg Recall: 0.636364,  count: 11
Region Avg IOU: 0.639657, Class: 1.000000, Obj: 0.188305, No Obj: 0.008358, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.632416, Class: 1.000000, Obj: 0.195704, No Obj: 0.008262, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.649878, Class: 1.000000, Obj: 0.248384, No Obj: 0.008911, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.609488, Class: 1.000000, Obj: 0.135753, No Obj: 0.008694, Avg Recall: 0.875000,  count: 8
102: 0.368064, 0.295509 avg, 0.001000 rate, 934.161072 seconds, 6528 images
Loaded: 0.000055 seconds
Region Avg IOU: 0.632289, Class: 1.000000, Obj: 0.100145, No Obj: 0.006912, Avg Recall: 0.625000,  count: 8
Region Avg IOU: 0.640102, Class: 1.000000, Obj: 0.131865, No Obj: 0.007346, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.686651, Class: 1.000000, Obj: 0.078864, No Obj: 0.007308, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.650147, Class: 1.000000, Obj: 0.141245, No Obj: 0.007896, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.693586, Class: 1.000000, Obj: 0.201208, No Obj: 0.008889, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.597732, Class: 1.000000, Obj: 0.188538, No Obj: 0.007033, Avg Recall: 1.000000,  count: 9
Region Avg IOU: 0.601624, Class: 1.000000, Obj: 0.157620, No Obj: 0.006813, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.527859, Class: 1.000000, Obj: 0.118578, No Obj: 0.007240, Avg Recall: 0.583333,  count: 12
103: 0.319020, 0.297860 avg, 0.001000 rate, 933.880310 seconds, 6592 images
Loaded: 0.000070 seconds
Region Avg IOU: 0.652756, Class: 1.000000, Obj: 0.150491, No Obj: 0.006900, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.559704, Class: 1.000000, Obj: 0.117521, No Obj: 0.006857, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.661993, Class: 1.000000, Obj: 0.100310, No Obj: 0.007276, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.616868, Class: 1.000000, Obj: 0.155978, No Obj: 0.007540, Avg Recall: 0.875000,  count: 8
Region Avg IOU: 0.609272, Class: 1.000000, Obj: 0.101161, No Obj: 0.007049, Avg Recall: 0.750000,  count: 8
Region Avg IOU: 0.706560, Class: 1.000000, Obj: 0.129159, No Obj: 0.007612, Avg Recall: 1.000000,  count: 8
Region Avg IOU: 0.578526, Class: 1.000000, Obj: 0.111814, No Obj: 0.006497, Avg Recall: 0.545455,  count: 11
Region Avg IOU: 0.666236, Class: 1.000000, Obj: 0.096877, No Obj: 0.006757, Avg Recall: 0.875000,  count: 8
104: 0.286160, 0.296690 avg, 0.001000 rate, 934.626770 seconds, 6656 images
Loaded: 0.000079 seconds
Region Avg IOU: 0.581765, Class: 1.000000, Obj: 0.045556, No Obj: 0.007185, Avg Recall: 0.818182,  count: 11
Region Avg IOU: 0.592564, Class: 1.000000, Obj: 0.088673, No Obj: 0.008038, Avg Recall: 0.625000,  count: 8
Region Avg IOU: 0.681468, Class: 1.000000, Obj: 0.096284, No Obj: 0.005918, Avg Recall: 0.875000,  count: 8
```

Figure 4. First Training Result

We can see the average loss for 99th and 100th iteration was 0.291857 and 0.287070, respectively. According to Darknet documentation, early stopping point occurs when average loss no longer decreases 0.060730[9]. Therefore, the difference of 0.004787 was much less than indicated threshold, which will cause some errors.

We also performed another training with 400 testing images and 100 validation datasets. However, even this time it only cost 1 day of training, the testing result was worse than the first training. Therefore, we will only present the testing results of the first training in the following section.

2.1.5 YOLO Testing

We tested several images using the obtained trained weight, and made comparisons with the pre-trained weights. Here is one example as shown in Figure 5 and 6. Command is formatted as:

```
./darknet detector test <obj.data> <cfg file> <weight file> <test image>
```

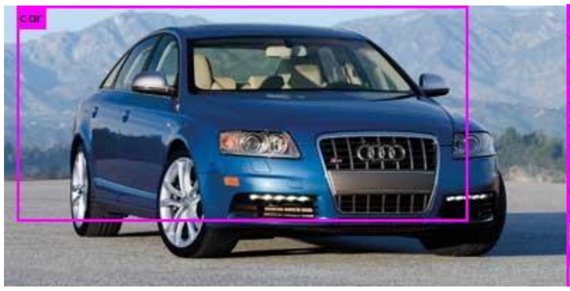


Figure 5. Prediction After Training

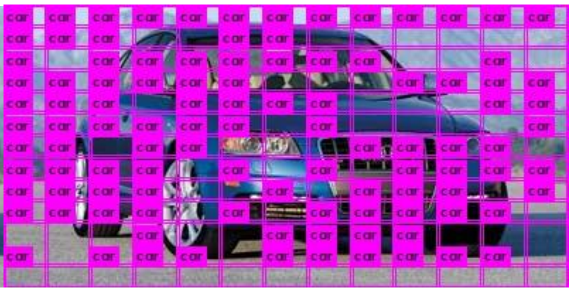


Figure 6.Prediction Before Training

We can see from the figures above that small number of training datasets improved the result significantly, providing the starting weight can only divide the image into 13x13 blocks and predict each element. However, insufficient data lead to low confidence of prediction, which reported only 25% confidence in Figure 5. From other test cases, the average confidence was around 7%, and sometimes multiple prediction bounding boxes appeared even if only one car was present.

To improve this, according to the Darknet documentation, we needed at least 2000 training images and 1000 validation images, and perform at least 1000 iterations to select the weight file with the best performance. Due to time limitations and device constraints, we were unable to achieve this in this project [9].

Additionally, in order to pass clean images without bounding box to the next phase, we modified the code so that the prediction area could be cropped and saved. The procedure is included in “README.md” in the darknet folder.

2.2 License Plate Detection and Character Recognition

2.2.1 Components

In order to create an effective license plate reading program, we needed to divide the process into separate components. First, the license plate had to be located within an image of a car. Next, the individual letters and numbers contained within the plate needed to be identified and cropped. Finally, the value of the cropped images of the individual characters had to be predicted.

2.2.2 License Plate Detection

The initial design of the license plate detection component was based on the assumptions that the input image would be a close-up picture of a vehicle with a North American style license plate. This component was developed using the Scikit-Image library, in addition to open source code by Femi Oladeji [27].

The input image was first converted to a binary image using Otsu’s method by creating a threshold. The connected components in the binary image are then located and labelled. These labelled regions have properties which can be iterated through. Using the regionprops() function of the Skimage library, the labelled regions were compared with certain criteria to see if they could be a license plate.

We tried many different combinations of criteria to determine if a region was a license plate, such as the region’s size relative to image size, and the height to width ratio. Unfortunately, it was difficult to develop a reliable set of criteria since there were many possible variances in the input images. If successful, this step yielded a region that contained the black and white license plate.

2.2.3 Segmentation

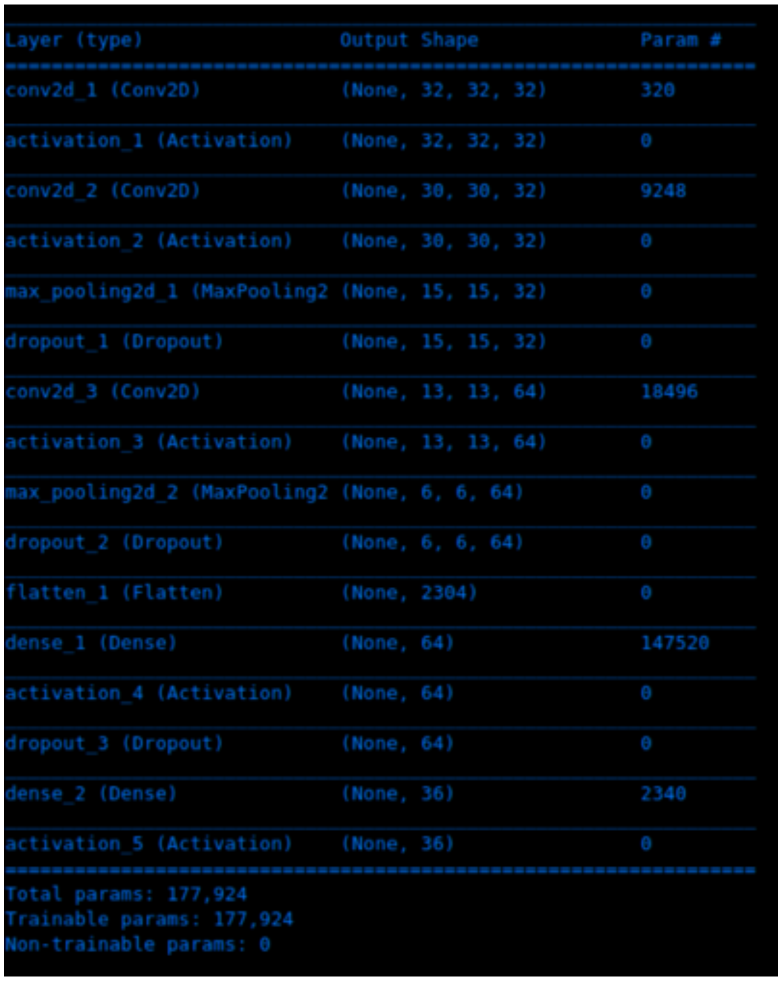
Locating the characters within the license plate was done in a similar fashion as the previous step. First, the region containing the plate was labelled. Then, the regions within the plate were tested against some criteria. In order to find the license plate characters, the criteria we used were minimum and maximum dimensions based on the plate size. Once located, the characters were resized to 32x32, and appended to a list to be used in the next step.

2.2.4 Prediction

Character recognition was achieved using deep learning. Specifically, we created a convolutional neural network using Keras and TensorFlow. Our first model, based on The Semicolon’s code only gave us 71% accuracy [29]. We subsequently changed to a model by Anuj Shah [28], which gave us much better results. The CNN model consisted of three twodimensional convolution layers, and multiple drop out and max-pooling layers. The model is summarized in figure 7. We chose to use the adam optimizer as our update procedure.

In order to train the network, we used a data set called Chars74k, which contained thousands of images consisting of letters and numbers [30]. We narrowed down the dataset to only black and white computer fonts of capital letters and numbers from 0 to 9.

At this point we ran into an issue regarding the competency of our hardware. Since we did not have access to a high-performance GPU, we had to train on a laptop CPU. During training, the sensors were reporting very high CPU temperatures. Due to the risk of overheating the CPU, we decided to constrain our training in a few different ways. Primarily, we wanted to keep training time at a minimum. Therefore, we decided to cap the number of epochs at 10. Another way to reduce training time, as well as memory usage, was to reduce the size of the training images. We decided that 32x32 images would be good enough to achieve the result we were expecting.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 32)        320
_________________________________________________________________
activation_1 (Activation)    (None, 32, 32, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 30, 30, 32)        9248
_________________________________________________________________
activation_2 (Activation)    (None, 30, 30, 32)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 15, 15, 32)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 15, 15, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 13, 64)        18496
_________________________________________________________________
activation_3 (Activation)    (None, 13, 13, 64)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                147520
_________________________________________________________________
activation_4 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 36)                2340
_________________________________________________________________
activation_5 (Activation)    (None, 36)                0
=================================================================
Total params: 177,924
Trainable params: 177,924
Non-trainable params: 0
```

Figure 7. Neural Network Summary

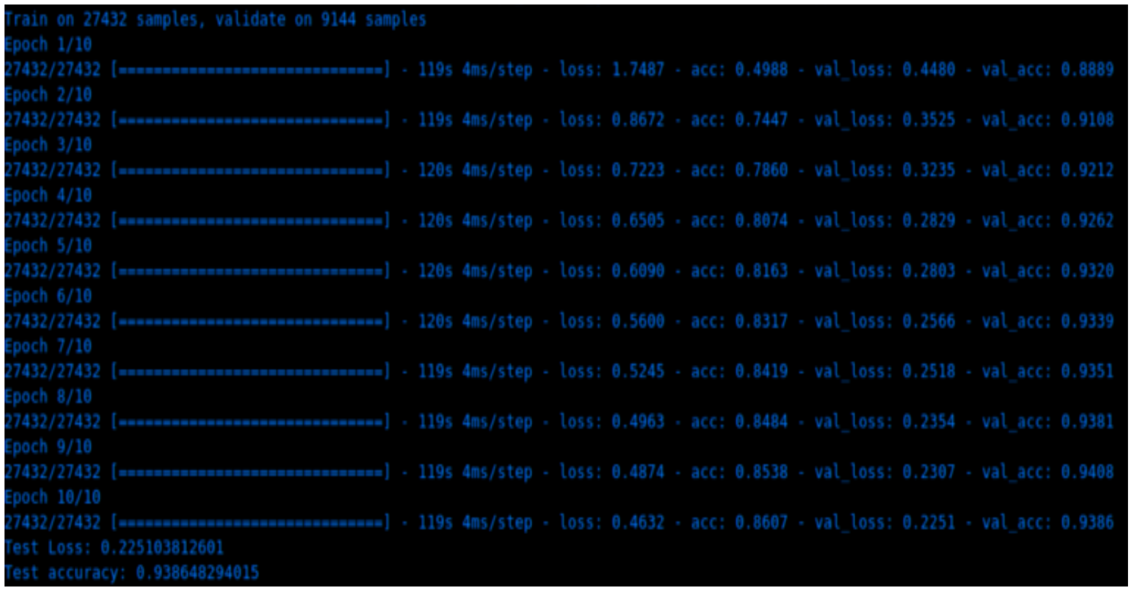
The newly trained model took about 2 minutes per epoch for a total of about 20 minutes. The reported accuracy was ~94% and a test loss was ~0.225. This model was saved into a .h5 file to be used in the prediction program.

The prediction program was initially designed to read the segmented letters from the previous segmentation step. This required saving images to numpy arrays, and expanding the array dimensions to satisfy Keras' input requirements. For TensorFlow, the input had to be the following:

[“number of images”, “Image dimension1”, “Image dimension2”, “Channels”]

Ultimately, this method was unsuccessful, since there were some errors in properly formatting the images from the previous step. However, we were able to create a prediction program that read images directly from a folder, processed them (resize and single channel) and output an array of predictions. The array was translated from numbered classes to the appropriate class names using a Python Dict type, and some basic logic/array manipulation.

Although the prediction program was fairly successful at reading many different types of character images, it was not perfect. We believe that the model could be improved by increasing the number of epochs, and using higher resolution images for training. Also, the fact that certain classes could be nearly identical (such as ‘O’ and ‘0’, or ‘1’ and ‘L’) could be another issue with our model.



```
Train on 27432 samples, validate on 9144 samples
Epoch 1/10
27432/27432 [==============================] - 119s 4ms/step - loss: 1.7487 - acc: 0.4988 - val_loss: 0.4480 - val_acc: 0.8889
Epoch 2/10
27432/27432 [==============================] - 119s 4ms/step - loss: 0.8672 - acc: 0.7447 - val_loss: 0.3525 - val_acc: 0.9108
Epoch 3/10
27432/27432 [==============================] - 120s 4ms/step - loss: 0.7223 - acc: 0.7860 - val_loss: 0.3235 - val_acc: 0.9212
Epoch 4/10
27432/27432 [==============================] - 120s 4ms/step - loss: 0.6505 - acc: 0.8074 - val_loss: 0.2829 - val_acc: 0.9262
Epoch 5/10
27432/27432 [==============================] - 120s 4ms/step - loss: 0.6090 - acc: 0.8163 - val_loss: 0.2803 - val_acc: 0.9320
Epoch 6/10
27432/27432 [==============================] - 120s 4ms/step - loss: 0.5600 - acc: 0.8317 - val_loss: 0.2566 - val_acc: 0.9339
Epoch 7/10
27432/27432 [==============================] - 119s 4ms/step - loss: 0.5245 - acc: 0.8419 - val_loss: 0.2518 - val_acc: 0.9351
Epoch 8/10
27432/27432 [==============================] - 119s 4ms/step - loss: 0.4963 - acc: 0.8484 - val_loss: 0.2354 - val_acc: 0.9381
Epoch 9/10
27432/27432 [==============================] - 119s 4ms/step - loss: 0.4874 - acc: 0.8538 - val_loss: 0.2307 - val_acc: 0.9408
Epoch 10/10
27432/27432 [==============================] - 119s 4ms/step - loss: 0.4632 - acc: 0.8607 - val_loss: 0.2251 - val_acc: 0.9386
Test Loss: 0.225103812601
Test accuracy: 0.938648294015
```

Figure 8. Test and accuracy training results

2.2.5 implementation

After training the “characters & numbers” and segmenting a license plate, figure 9 shows the connection between both tasks at the testing step.

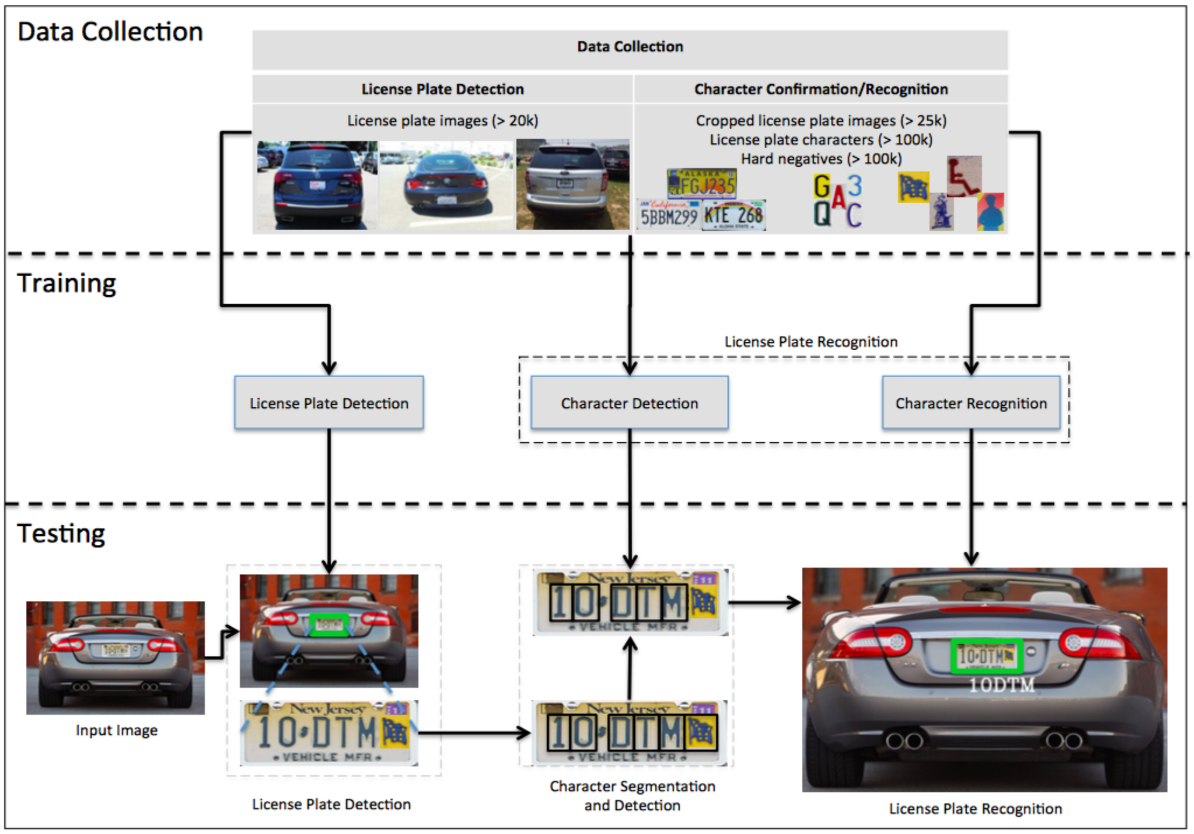

Data Collection
Data Collection
License Plate Detection
Character Confirmation/Recognition
License plate images (> 20k)
Cropped license plate images (> 25k)
License plate characters (> 100k)
Hard negatives (> 100k)
Training
License Plate Recognition
License Plate Detection
Character Detection
Character Recognition
Testing
Input Image
License Plate Detection
Character Segmentation and Detection
License Plate Recognition
10DTM

Figure 9 .Merge both character segmentation and detection algorithms[19]

3. Contribution of the work

As for work contribution, Arlene and Ricky did the first part, which is the car detection. Toky and Karamveer did the license plate recognition. Due to the time limitation, all of us are focusing our own part. Therefore, we did not write a single main file to integrate two parts. But after running the vehicle detection, the plate recognition part can directly treat the output from the first part as the input.

4. Result

Although integration of all the components was not realized, we were moderately successful with both our vehicle detection and character recognition. For vehicle detection, we were able to predict vehicle with up to 25% confidence, and our training showed a final loss of 0.287. For character recognition, our model had a final test loss of 0.225 and claimed an accuracy of 93%. It worked well for most fonts that we tested, but it is apparent that further work is required to improve it.

5. Conclusion and Summary

In summary, we were able to understand the architecture of the convolutional neural network (CNN) and its training procedure and understand the whole procedure of YOLO custom model. furthermore, we were able increase the prediction precision by training a small dataset. The aspiration is detecting foreign license plates with the trained international dataset. In conclusion, there are multiples of applications in computer vision that deep learning could be used for such.

6. Reference

[1] J. (2017, July 20). Created vehicle detection pipeline with two approaches: (1) deep neural networks (YOLO framework) and (2) support vector machines (OpenCV HOG). Retrieved November 29, 2017, from https://github.com/JunshengFu/vehicle-detection
[2] Unknown. (n.d.). Github source code from https://github.com/sunshineatnoon/Paper-Collection/blob/master/YOLO.md
[3] Trieu. (n.d.), from https://github.com/thtrieu/darkflow
[4] Krause, J. Deng, J. Stark, M. and Li, F. F. (n.d.). Collecting a Large-Scale Dataset of Fine-Grained Cars, from http://ai.stanford.edu/~jkrause/papers/fgvc13.pdf
[5] Earl, M. (n.d.). Number plate recognition with Tensorflow. Retrieved November 29, 2017, from https://matthewearl.github.io/2016/05/06/cnn-anpr/
[6] Unknown. (n.d.). KITTI Vision Benchmark Suite. Retrieved November 29, 2017, from http://www.cvlibs.net/datasets/kitti/
[7] Tijtgat, N. (2017, June 07). How to train YOLOv2 to detect custom objects. Retrieved November 29, 2017, from https://timebutt.github.io/static/how-to-train-yolov2-to-detect-custom-objects/
[8] J. (2017, July 17). How to train YOLOv2 on custom dataset. Retrieved November 29, 2017, from https://jumabek.wordpress.com/2017/03/04/how-to-train-yolov2-on-costum-dataset/
[9] Unknown . (2017, November 28). AlexeyAB/darknet. Retrieved November 29, 2017, from https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects
[10] Redmon, J. (n.d.). YOLO: Real-Time Object Detection. Retrieved November 29, 2017, from https://pjreddie.com/darknet/yolo/
[11] P. (2017, November 26). Convolutional Neural Networks. Retrieved November 29, 2017, from https://github.com/pjreddie/darknet
[12] Unknown. (n.d.). Generate List of Numbers – Text Mechanic™. Retrieved November 29, 2017, from http://textmechanic.com/text-tools/numeration-tools/generate-list-numbers/
[13] Oladeji, F. (2017, August 03). Developing a License Plate Recognition System with Machine Learning in Python. Retrieved November 29, 2017, from https://blog.devcenter.co/developing-a-license-plate-recognition-system-with-machine-learning-in-python-787833569ccd
[14]Unknown . (2017, May 12). Tutorial on CNN implementation for own data set in keras (TF & Theano backend)-part-1. Retrieved November 29, 2017, from https://www.youtube.com/watch?v=u8BW_fl6WRc&t=1468s
[15] Mallick, S. (2017, February 13). Object Tracking using OpenCV (C /Python). Retrieved November 29, 2017, from http://www.learnopencv.com/object-tracking-using-opencv-cpp-python/
[16] U. (2017, July 28). Experiencor/self-driving-toy-car. Retrieved November 29, 2017, from https://github.com/experiencor/self-driving-toy-car
[17] Kazakov, I. (2017, May 14). Vehicle Detection and Tracking – Towards Data Science. Retrieved November 29, 2017, from https://medium.com/towards-data-science/vehicle-detection-and-tracking-44b851d70508
[18] Number plate recognition with Tensorflow. (n.d.). Retrieved November 29, 2017, from https://matthewearl.github.io/2016/05/06/cnn-anpr/
[19] Farfade, S. S., Saberian, M., & Li, L. (2015, March 4). Multi-view Face Detection Using Deep Convolutional Neural Networks. Retrieved from https://arxiv.org/pdf/1502.02766v2.pdf
[20] S., H., & H. (n.d.). Deep-learning-based license plate detection method using vehicle region extraction. Retrieved November 29, 2017, from http://ieeexplore.ieee.org/document/7990396/?reload=true
[21] Vehicle license plate recognition using Convolutional Neural Network trained with mnist data. (n.d.). Retrieved November 29, 2017, from https://stats.stackexchange.com/questions/165083/vehicle-license-plate-recognition-using-convolutional-neural-network-trained-wit

[22] Masood, S. Z., Shu, G., Dehghan, A., & Ortiz, E. G. (2017, March 28). License Plate Detection and Recognition Using Deeply Learned Convolutional Neural Networks. Retrieved November 29, 2017, from https://arxiv.org/pdf/1703.07330.pdf
[23] Unknown . (2017, September 12). A simple tool for labeling object bounding boxes in images. Retrieved November 29, 2017, from https://github.com/puzzledqs/BBox-Label-Tool
[24] Zhou, Y., Nejati, H., Do, T., Cheung, N., & Cheah, L. (2016, August 7). Image-based Vehicle Analysis using Deep Neural Network: A Systematic Study. Retrieved October 15, 2017, from https://arxiv.org/pdf/1601.01145.pdf
[25] Jonathan Krause, Michael Stark, Jia Deng, Li Fei-Fei, 3D Object Representations for Fine-Grained Categorization, 4th IEEE Workshop on 3D Representation and Recognition, at ICCV 2013 (3dRR-13). Sydney, Australia. Dec. 8, 2013. Dataset retrieved from http://ai.stanford.edu/~jkrause/cars/car_dataset.html
[26] Unknown (n.d.), from https://github.com/puzzledqs/BBox-Label-Tool
[27] F. Oladeji, "Developing a License Plate Recognition System with Machine Learning in Python," *Devcenter*, 03-Aug-2017. [Online]. Available: https://blog.devcenter.co/developing-a-license-plate-recognition-system-with-machine-learning-in-python-787833569ccd. [Accessed: 01-Nov-2017].
[28] Shah, A. (2017, August 30). own_data_cnn_implementation_keras**.** Retrieved November 15, 2017, from https://github.com/anujshah1003/own_data_cnn_implementation_keras/blob/master/custom_data_cnn.py
[29] The Semicolon. (2017, October 29). Deep Learning with Keras**.** Retrieved November 13, 2017, from https://github.com/shreyans29/thesemicolon
[30] The Semicolon. (2015, October 15). The Chars74K image dataset - Character Recognition in Natural Images**.** Retrieved November 13, 2017, from http://www.ee.surrey.ac.uk/CVSSP/demos/chars74k/