Deep Learning Based (CNN) Real-time Dynamic Vehicle Detection

NEU Software Engineering (International English)
Yuanshi Li

1. Theoretical introduction

1.1 What is convolution & Why we use convolution

(1) Convolution operation

Convolution is a special linear operation, a mathematical operation on two real-valued functions. As we all known, the Kalman filter which was introduced in the GNSS INS Integration part uses one-dimensional discrete convolution operation. One dimensional discrete convolution formula is as following:

$$s(i) = (h * w)(i) = \sum_{j=-\infty}^{\infty} h(j)w(i-j)$$

i is the operation state, j is the distance to i state. h is input, w is kernel function and s is the feature map. In terms of convolutional neural network, the first function h is called input, the second function w is called kernel function, and the output s is called feature map.

Two-dimensional convolution operation formula can be shown as following:

$$S(m,n) = (I * K)(m,n) = \sum_i \sum_j I(i,j)K(m-i,n-j)$$

(m,n) is the pixel location and (i,j) is the scope of the consideration. In a more intuitive form, the two-dimensional convolution is as follows:

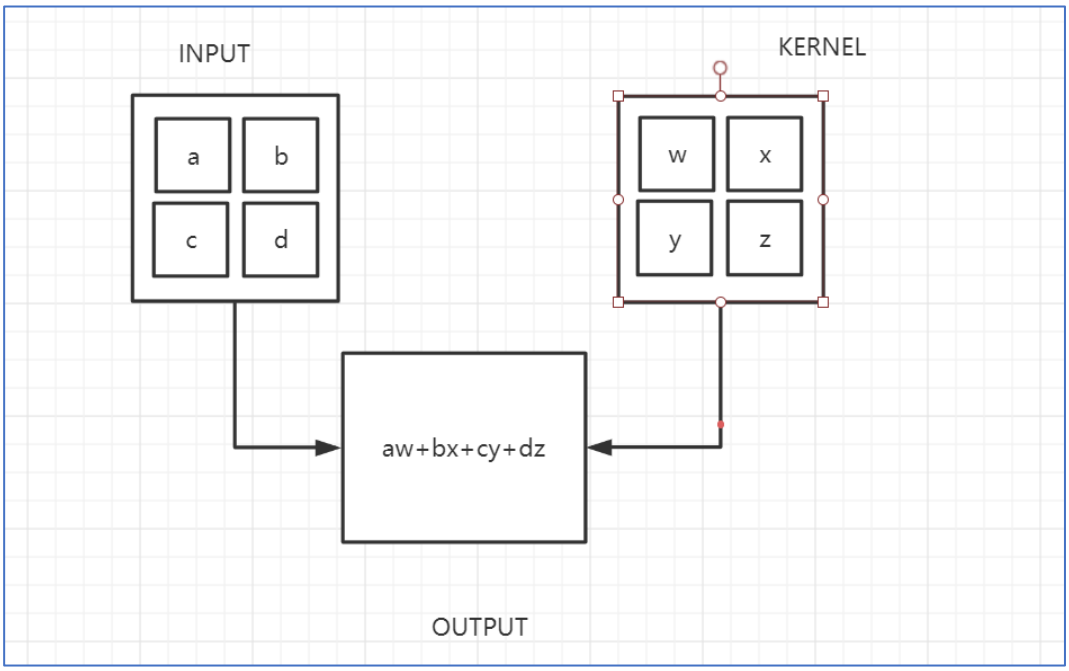

Fig 1.1.1 The workflow of the two-dimensional convolution

(2) Why we use convolution

We use convolution to improve machine learning systems. For a common fully connected network, the nodes between layers are fully connected. But for the convolutional network, the nodes in the next layer are only related to the nodes affected by the convolutional kernel.

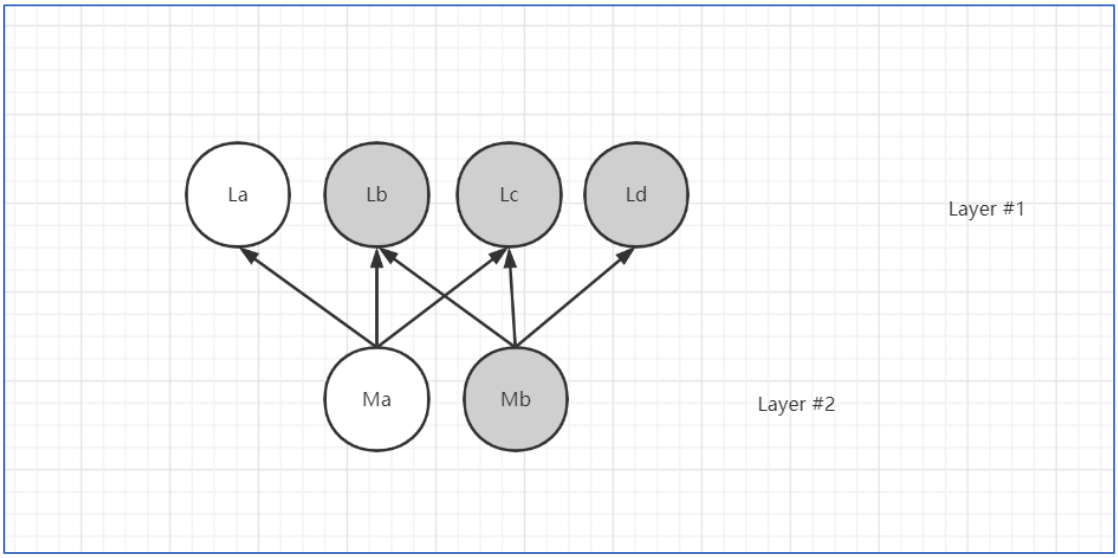

Fig 1.1.2 The principle of Sparse Interactions

In this diagram, Mb is only associated with Lb, Lc and Ld. That is why we called it sparse interactions and this is the first advantage that could be used to train the classifier model.

One of the intuitive benefits of using sparse connections is that the network has fewer parameters. Let's take a gray scale of 200×200 for example, when it is fed into a fully connected neural network, as shown below:

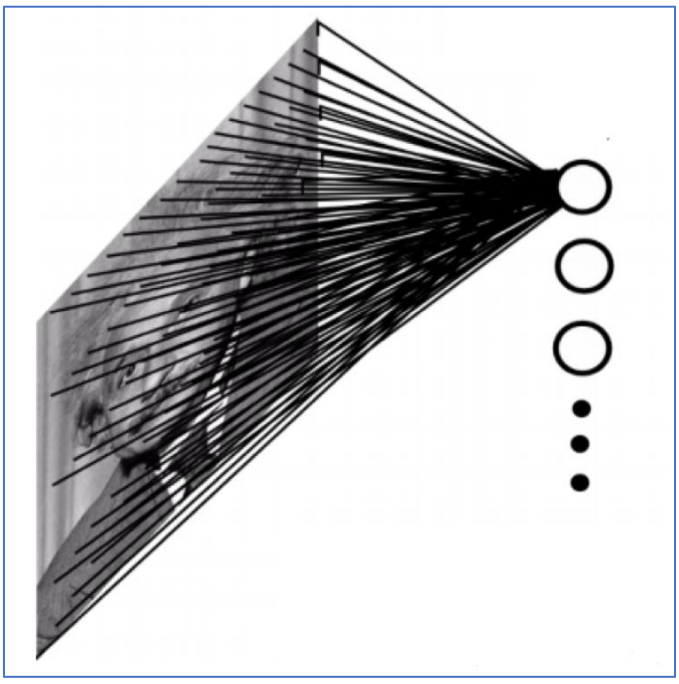

Fig 1.1.3 The picture in the neural network

Assuming that the first hidden layer of the network has 40000 neurons (40000 hidden layer nodes are appropriate for an input sample of 40000 dimensions), then the light layer of the network has close to 2 billion parameters. Such model training is very computationally intensive and requires a large amount of storage space.

But for the convolutional network, the situation is as follows:

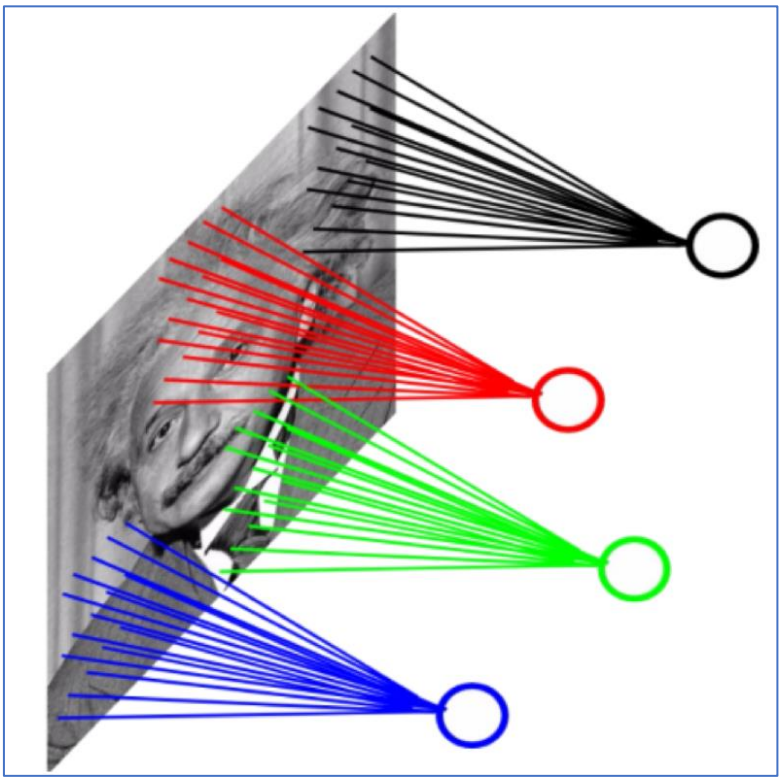

Fig 1.1.4 The picture in the convolutional neural network

Here we still use 40000 hidden layer neurons, and the size of our convolution kernel (also known as Filter) is 10×10. Such a convolution has only about 4 million parameters, which is much smaller than the number of fully connected network.

The next advantage is parameter sharing. The convolution kernel is actually the parameter of the convolutional network. The convolution kernel slides the window on the input image, which means that the pixels of the input image share this set of parameters, as shown in the following figure:

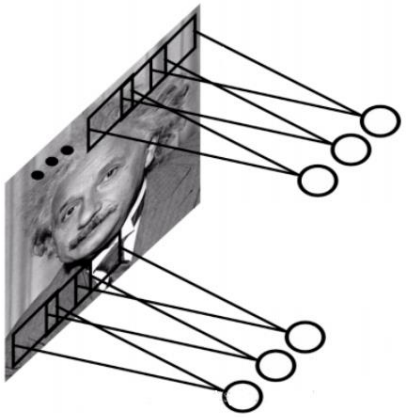

Fig 1.1.5 parameter sharing

The parameter sharing in the convolutional network makes us only need to learn a parameter set instead of a single parameter set for each pixel, which greatly reduces the storage space required for the model.

The last advantage is equivariant representations. The formula of equal variability:

$$f(g(x)) = g(f(x))$$

These two properties can help simplify the input parameters of the CNN and improve the efficiency of machine learning system.

1.2 CNN workflow

The whole system works with the classical deep learning model CNN (convolutional neural network). The CNN workflow is as following:

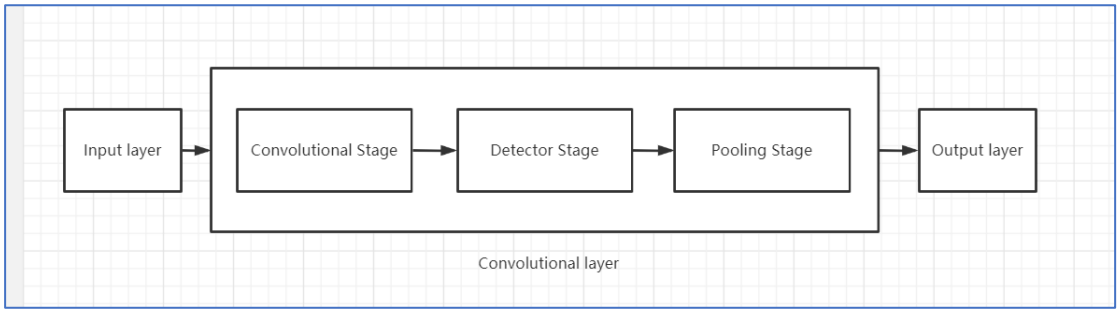

Fig 1.2.1 CNN workflow

2. YOLO based real-time dynamic vehicle detection

2.1 YOLO

YOLO (you only look once) is a target detection model. YOLO is one of the deep learning target detection algorithms based on regression method. We will introduce the deep learning target detection method based on the regression method -- YOLO, and use the tiny version of YOLO to realize the vehicle detection DEMO.

YOLO sees target detection as a regression problem, and the workflow of a well-trained network is very simple, as shown in the following figure:

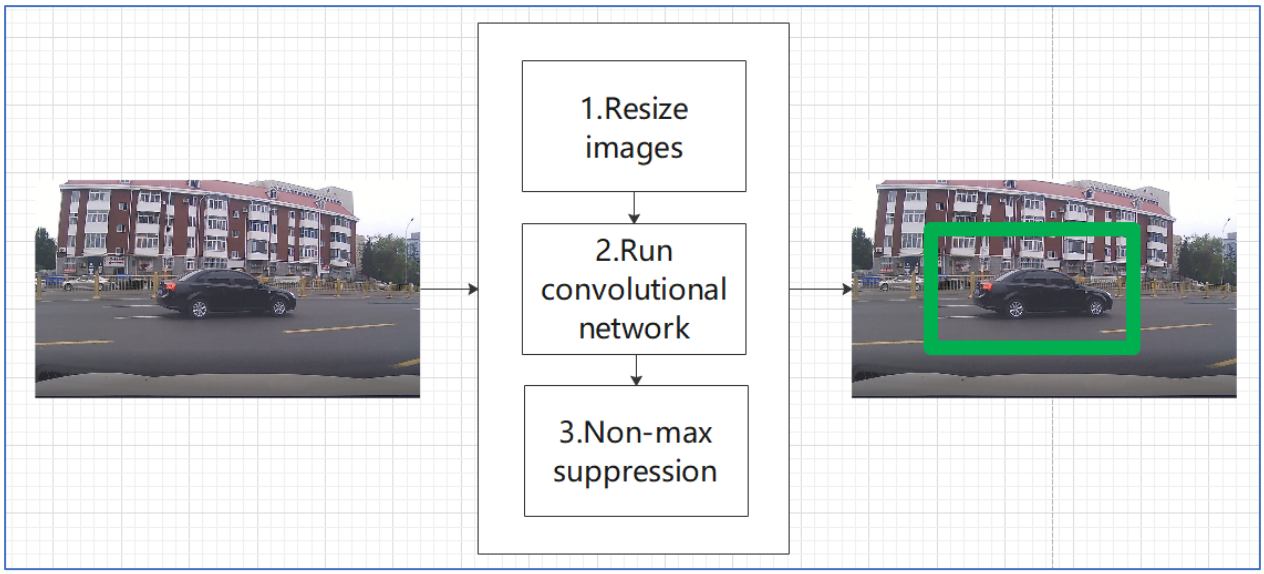

Fig 2.1.1 YOLO workflow (same to the CNN workflow but do regression tasks)

As shown in the figure, as an end-to-end network, the original image is input, and the output is the location of the target, its category and the corresponding confidence probability. Unlike traditional sliding window detection algorithms, YOLO uses the entire image as input in both training and application stages. YOLO's specific network structure is as follows:

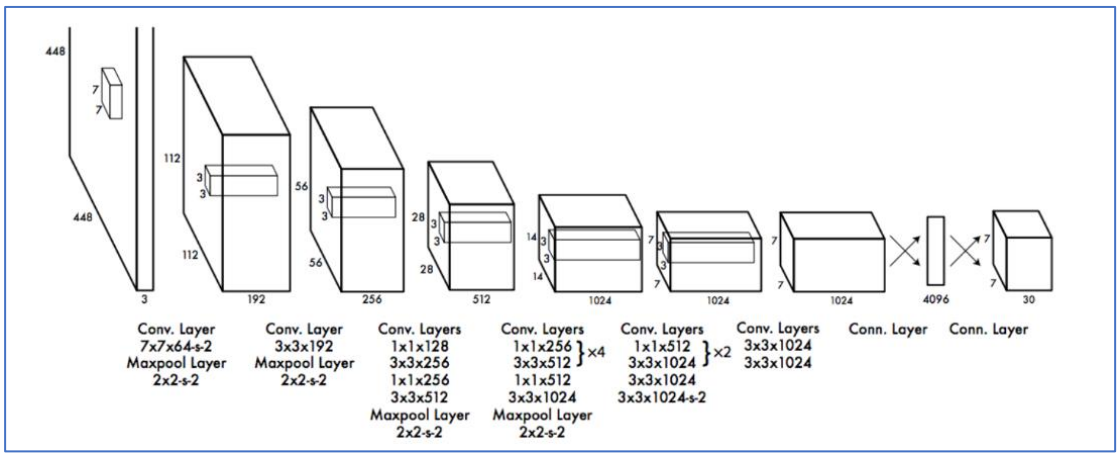

Fig 2.1.2 The specific network structure of YOLO

2.2 Pre-training classification network

First, a classification network with input of 224×224 is trained on the ImageNet dataset using the first 20 convolutional layers + an average pooling layer + a full connection layer.

2.3 Training detection network

After the pre-trained 20 convolutional layers, we add 4 convolutional layers and 2 fully linked layers. The input is 448 × 448 and the output is a 7 × 7 × 30 tensor which means the input images will be fragmented into 7 × 7 small pieces.

Then what we need to focus on is the regression prediction of these 7 × 7 grids.

Each small area contains three pieces of information: The center coordinates, the height and width of the area and the confidence.

Each grid needs to predict two pieces of images so we have (3+2) * 2 = 10 outputs. According to YOLO, we have to predict 20 classes. The expectable output of each grid is 30 values.

Our final layer predicts both class probabilities and bounding box coordinates. We normalize the bounding box width and height by the image width and height so that they fall between 0 and 1. We parametrize the bounding box x and y coordinates to be offsets of a particular grid cell location so they are also bounded between 0 and 1.

We use a linear activation function for the final layer and all other layers use the following leaky rectified linear activation:

$$\phi(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases}$$

Fig 2.3.1 Linear activation

We optimize for sum-squared error in the output of our model. We use sum-squared error because it is easy to optimize, however it does not perfectly align with our goal of maximizing average precision. It weights localization error equally with classification error which may not be ideal. Also, in every image many grid cells do not contain any object. This pushes the "confidence" scores of those cells towards zero, often overpowering the gradient from cells that do contain objects. This can lead to model instability, causing training to diverge early on.

2.4 Loss function

We have got 30 output values from training the detection network.

To get a good return on these 30 values, the loss function design must be balanced in the center coordinates, the height and width of the area and the confidence.

A search of YOLO's papers shows that we need to use the multi-part loss function to balance the three attributes.

YOLO uses the following function as a loss function to detect the network:

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(x_i - \hat{x}_i\right)^2 + \left(y_i - \hat{y}_i\right)^2$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c)\right)^2 \quad (3)$$

where $\mathbb{1}_i^{\text{obj}}$ denotes if object appears in cell i and $\mathbb{1}_{ij}^{\text{obj}}$ denotes that the jth bounding box predictor in cell i is "responsible" for that prediction.

Fig 2.4.1 The loss function of YOLO

Actually, it is just essentially a regression equation of the specific YOLO model which belongs to CNN.

2.5 Final test

In final test step, the category information of each grid prediction is multiplied by the confidence of bounding box (each piece of these segmented image) prediction so that we get the class-specific confidence score.

If we do this for each grid of the entire image, we can get $7 \times 7 \times 2 = 98$ bounding boxes, which contain both coordinate and category information.

After the class-specific confidence score of each box is obtained, the threshold value is set, boxes with low scores are filtered out, and rest boxes are processed by NMS to obtain the final detection result.

3. Vehicle detection model based on YOLO

Due to the high real-time requirement of vehicle detection, we use a simplified version of YOLO: Fast YOLO. This model uses a simple 9-layer convolution instead of the original 24-layer convolution. It sacrifices some accuracy and is faster in processing, improving from YOLO's 45fps to 155fps. To meet the needs of real-time target detection.

3.1 Implement fast YOLO network structure using Keras

```
In [28]: # this is because we use tensorflow as the backend

         # keras.backend.set_image_dim_ordering('th')

In [29]: model = Sequential()
         model.add(Convolution2D(16, 3, 3,input_shape=(3,448,448),border_mode='same',subsample=(1,1)))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Convolution2D(32,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))
         model.add(Convolution2D(64,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))
         model.add(Convolution2D(128,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))
         model.add(Convolution2D(256,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))
         model.add(Convolution2D(512,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(MaxPooling2D(pool_size=(2, 2),border_mode='valid'))
         model.add(Convolution2D(1024,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(Convolution2D(1024,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(Convolution2D(1024,3,3 ,border_mode='same'))
         model.add(LeakyReLU(alpha=0.1))
         model.add(Flatten())
         model.add(Dense(256))
         model.add(Dense(4096))
         model.add(LeakyReLU(alpha=0.1))
         model.add(Dense(1470))

In [30]: model.summary()
```

Fig 3.1.1 Using Keras to make YOLO model

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_19 (Conv2D)           (None, 16, 448, 448)      448
_________________________________________________________________
leaky_re_lu_21 (LeakyReLU)   (None, 16, 448, 448)      0
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 16, 224, 224)      0
_________________________________________________________________
conv2d_20 (Conv2D)           (None, 32, 224, 224)      4640
_________________________________________________________________
leaky_re_lu_22 (LeakyReLU)   (None, 32, 224, 224)      0
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 32, 112, 112)      0
_________________________________________________________________
conv2d_21 (Conv2D)           (None, 64, 112, 112)      18496
_________________________________________________________________
leaky_re_lu_23 (LeakyReLU)   (None, 64, 112, 112)      0
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 64, 56, 56)        0
_________________________________________________________________
conv2d_22 (Conv2D)           (None, 128, 56, 56)       73856
_________________________________________________________________
leaky_re_lu_24 (LeakyReLU)   (None, 128, 56, 56)       0
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 128, 28, 28)       0
_________________________________________________________________
conv2d_23 (Conv2D)           (None, 256, 28, 28)       295168
_________________________________________________________________
leaky_re_lu_25 (LeakyReLU)   (None, 256, 28, 28)       0
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 256, 14, 14)       0
_________________________________________________________________
conv2d_24 (Conv2D)           (None, 512, 14, 14)       1180160
_________________________________________________________________
leaky_re_lu_26 (LeakyReLU)   (None, 512, 14, 14)       0
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 512, 7, 7)         0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 1024, 7, 7)        4719616
_________________________________________________________________
leaky_re_lu_27 (LeakyReLU)   (None, 1024, 7, 7)        0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 1024, 7, 7)        9438208
_________________________________________________________________
leaky_re_lu_28 (LeakyReLU)   (None, 1024, 7, 7)        0
_________________________________________________________________
conv2d_27 (Conv2D)           (None, 1024, 7, 7)        9438208
_________________________________________________________________
leaky_re_lu_29 (LeakyReLU)   (None, 1024, 7, 7)        0
_________________________________________________________________
flatten_3 (Flatten)          (None, 50176)             0
_________________________________________________________________
dense_7 (Dense)              (None, 256)               12845312
_________________________________________________________________
dense_8 (Dense)              (None, 4096)              1052672
_________________________________________________________________
leaky_re_lu_30 (LeakyReLU)   (None, 4096)              0
_________________________________________________________________
dense_9 (Dense)              (None, 1470)              6022590
=================================================================
Total params: 45,089,374
Trainable params: 45,089,374
Non-trainable params: 0
_________________________________________________________________
```

Fig 3.1.2 The trainable params

3.2 Load parameter files into the network

```python
def load_weights(model, yolo_weight_file):
    tiny_data = np.fromfile(yolo_weight_file, np.float32)[4:]

    index = 0
    for layer in model.layers:
        weights = layer.get_weights()
        if len(weights) > 0:
            filter_shape, bias_shape = [w.shape for w in weights]
            if len(filter_shape) > 2:  # For convolutional layers
                filter_shape_i = filter_shape[::-1]
                bias_weight = tiny_data[index:index + np.prod(bias_shape)].reshape(bias_shape)
                index += np.prod(bias_shape)
                filter_weight = tiny_data[index:index + np.prod(filter_shape_i)].reshape(filter_shape_i)
                filter_weight = np.transpose(filter_weight, (2, 3, 1, 0))
                index += np.prod(filter_shape)
                layer.set_weights([filter_weight, bias_weight])
            else:  # For regular hidden layers
                bias_weight = tiny_data[index:index + np.prod(bias_shape)].reshape(bias_shape)
                index += np.prod(bias_shape)
                filter_weight = tiny_data[index:index + np.prod(filter_shape)].reshape(filter_shape)
                index += np.prod(filter_shape)
                layer.set_weights([filter_weight, bias_weight])
```

Fig 3.2.1 Load trained weight file to the model

3.3 The detection results were extracted from the output of YOLO network

```python
def yolo_net_out_to_car_boxes(net_out, threshold=0.2, sqrt=1.8, C=20, B=2, S=7):
    class_num = 6
    boxes = []
    SS = S * S  # number of grid cells
    prob_size = SS * C  # class probabilities
    conf_size = SS * B  # confidences for each grid cell

    probs = net_out[0: prob_size]
    confs = net_out[prob_size: (prob_size + conf_size)]
    cords = net_out[(prob_size + conf_size):]
    probs = probs.reshape([SS, C])
    confs = confs.reshape([SS, B])
    cords = cords.reshape([SS, B, 4])

    for grid in range(SS):
        for b in range(B):
            bx = Box()
            bx.c = confs[grid, b]
            bx.x = (cords[grid, b, 0] + grid % S) / S
            bx.y = (cords[grid, b, 1] + grid // S) / S
            bx.w = cords[grid, b, 2] ** sqrt
            bx.h = cords[grid, b, 3] ** sqrt
            p = probs[grid, :] * bx.c

            if p[class_num] >= threshold:
                bx.prob = p[class_num]
                boxes.append(bx)

    # combine boxes that are overlap
    boxes.sort(key=lambda b: b.prob, reverse=True)
    for i in range(len(boxes)):
        boxi = boxes[i]
        if boxi.prob == 0: continue
        for j in range(i + 1, len(boxes)):
            boxj = boxes[j]
            if box_iou(boxi, boxj) >= .4:
                boxes[j].prob = 0.
    boxes = [b for b in boxes if b.prob > 0.]

    return boxes
```

Fig 3.3.1 Extract result from YOLO network

4. Apply to images and videos

4.1 Apply to images

4.1.1 Interpolate vector and generate boxes

apply the model to images

```
In [32]: imagePath = './test_images/test1.jpg'
         image = plt.imread(imagePath)
         image_crop = image[300:650,500:,:]
         resized = cv2.resize(image_crop,(448,448))

In [33]: batch = np.transpose(resized,(2,0,1))
         batch = 2*(batch/255.) - 1
         batch = np.expand_dims(batch, axis=0)
         out = model.predict(batch)
```

interpolate the vector out from the neural network, generate the boxes

```
In [34]: boxes = yolo_net_out_to_car_boxes(out[0], threshold = 0.17)
```

visualize the box on the original image

```
In [35]: f,(ax1,ax2) = plt.subplots(1,2,figsize=(16,6))
         ax1.imshow(image)
         ax2.imshow(draw_box(boxes,plt.imread(imagePath),[[500,1280],[300,650]]))
```

Fig 4.1.1.1 Interpolate vector and generate boxes

4.1.2 Output (single image test result)

```
Out[35]: <matplotlib.image.AxesImage at 0x20738caa688>
```

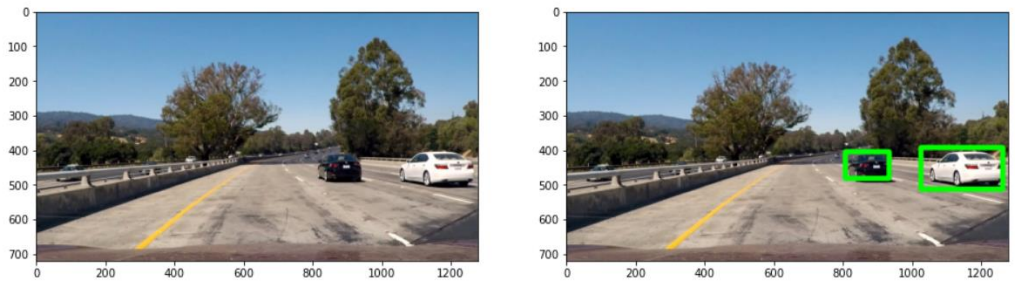

Fig 4.1.2.1 Output

4.1.3 Test about more images

other tests on images

```
In [36]: images = [plt.imread(file) for file in glob.glob('./test_images/*.jpg')]
         batch = np.array([np.transpose(cv2.resize(image[300:650,500:,:],(448,448)),(2,0,1))
                           for image in images])
         batch = 2*(batch/255.) - 1
         out = model.predict(batch)
         f,((ax1,ax2),(ax3,ax4),(ax5,ax6)) = plt.subplots(3,2,figsize=(11,10))
         for i,ax in zip(range(len(batch)),[ax1,ax2,ax3,ax4,ax5,ax6]):
             boxes = yolo_net_out_to_car_boxes(out[i], threshold = 0.17)
             ax.imshow(draw_box(boxes,images[i],[[500,1280],[300,650]]))
```

Fig 4.1.3.1 Output

4.1.4 Output (more image test result)

Fig 4.1.4.1 Output

4.2 Apply to images

We use the moviepy instrument to extract video files frame by frame. Then do YOLO detection to each frame.

apply to the video

```
In [37]: def frame_func(image):
             crop = image[:,80:438,:]
             resized = cv2.resize(crop,(448,448))
             batch = np.array([resized[:,:,0],resized[:,:,1],resized[:,:,2]])
             batch = 2*(batch/255.) - 1
             batch = np.expand_dims(batch, axis=0)
             out = model.predict(batch)
             boxes = yolo_net_out_to_car_boxes(out[0], threshold = 0.17)
             return draw_box(boxes,image,[[80,438],[0,288]])

In [38]: project_video_output = './movietest.mp4'
         clip1 = VideoFileClip("./movietest.avi")

In [39]: lane_clip = clip1.fl_image(frame_func)
         %time lane_clip.write_videofile(project_video_output, audio=False)

         t:  1%|         | 2/350 [00:00<00:26, 13.28it/s, now=None]

         Moviepy - Building video ./movietest.mp4.
         Moviepy - Writing video ./movietest.mp4



         Moviepy - Done !
         Moviepy - video ready ./movietest.mp4
         Wall time: 46.8 s
```

Fig 4.2.1 Segment and detect

The test result screenshot is as following:

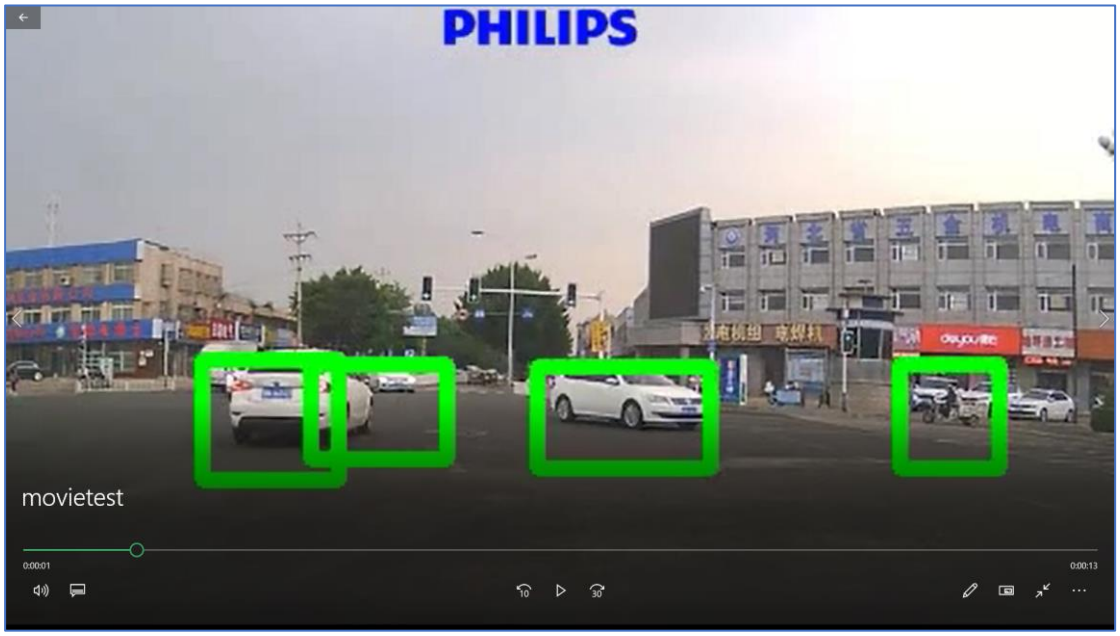

Fig 4.2.2 Result #1

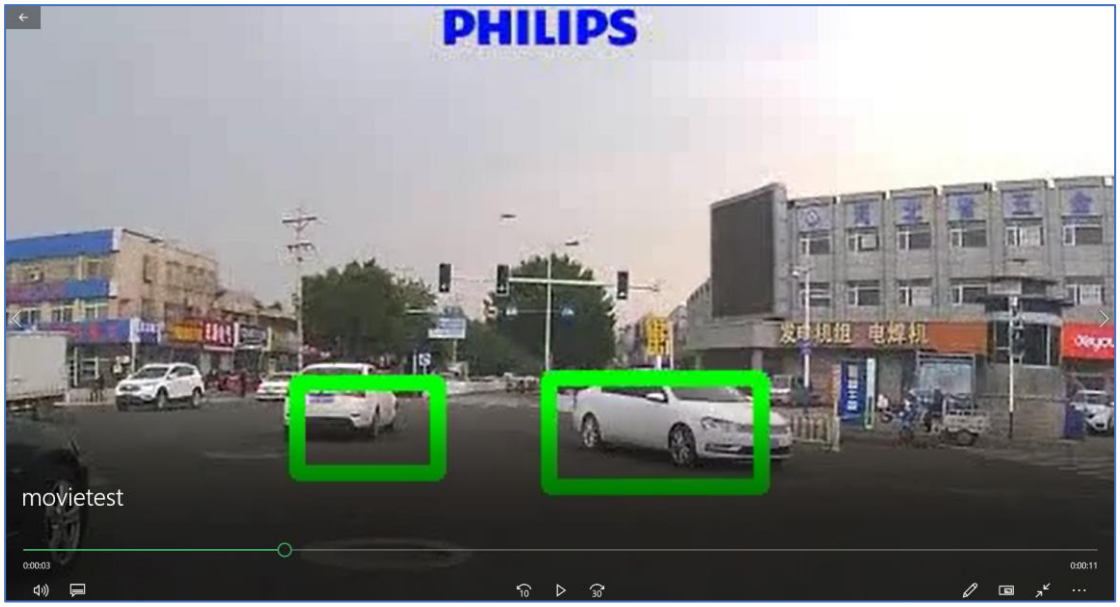

Fig 4.2.3 Result #2

Fig 4.2.4 Result #3

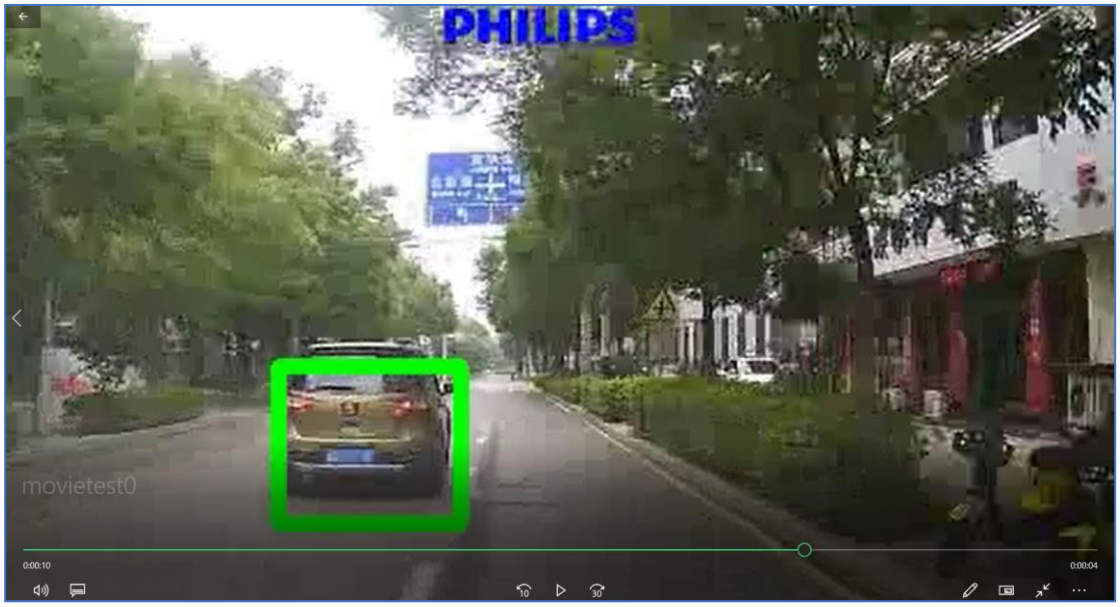

Fig 4.2.5 Result #4

5. Research results and expectations

5.1 Limitations and deficiencies

1. Lack of tracking detection model training to the current model.
2. The yolo_tiny model only has 9 convolution layers so that some specific value will not be fully trained.
3. The limitation of YOLO itself which has a strong spatial constraint on bounding box predictions so that it struggles to generalize to objects in new or unusual aspect ratios or configurations.
4. Environmental noise problem.

5.2 Application in the field of autonomous driving

Image recognition and dynamic obstacle recognition belong to the category of computer vision, but the effect of traditional methods is not ideal, especially in the field of autonomous driving is more difficult. With the development of the field of artificial intelligence in recent years, the application of machine learning methods in the field of computer vision has achieved remarkable results, greatly improving the timeliness and accuracy of prediction. Therefore, with the development of technology, these new methods will be more widely applied in the field of autonomous driving.

This algorithm can be used in the autonomous driving system' detection part. Its input could be the images which are taken by the camera on the vehicles. The output will be used in driving perception that prevent collision.

6. References

[1] M. B. Blaschko and C. H. Lampert. Learning to localize objects with structured output regression. In Computer Vision– ECCV 2008, pages 2–15. Springer, 2008. 4

[2] L. Bourdev and J. Malik. Poselets: Body part detectors trained using 3d human pose annotations. In International Conference on Computer Vision (ICCV), 2009. 8

[3] H. Cai, Q. Wu, T. Corradi, and P. Hall. The crossdepiction problem: Computer vision algorithms for recognising objects in artwork and in photographs. arXiv preprint arXiv:1505.00110, 2015. 7

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on, volume 1, pages 886–893. IEEE, 2005. 4, 8

[5] T. Dean, M. Ruzon, M. Segal, J. Shlens, S. Vijayanarasimhan, J. Yagnik, et al. Fast, accurate detection of 100,000 object classes on a single machine. In Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on, pages 1814–1821. IEEE, 2013. 5

[6] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. arXiv preprint arXiv:1310.1531, 2013. 4

[7] J. Dong, Q. Chen, S. Yan, and A. Yuille. Towards unified object detection and semantic segmentation. In Computer Vision–ECCV 2014, pages 299–314. Springer, 2014. 7

[8] D. Erhan, C. Szegedy, A. Toshev, and D. Anguelov. Scalable object detection using deep neural networks. In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, pages 2155–2162. IEEE, 2014. 5, 6

[9] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. International Journal of Computer Vision, 111(1):98–136, Jan. 2015. 2

[10] S. Ren, K. He, R. B. Girshick, X. Zhang, and J. Sun. Object detection networks on convolutional feature maps. CoRR, abs/1504.06066, 2015. 3, 7

[11] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision (IJCV), 2015. 3

[12] M. A. Sadeghi and D. Forsyth. 30hz object detection with dpm v5. In Computer Vision–ECCV 2014, pages 65–79. Springer, 2014. 5, 6