Vehicle Detection Project

The code that implemented and tested the algorithm is available in a python notebook file in the repo, named as "vehicle_detect_pj.ipynb"

Algorithm Used: You Only Look Once (YOLO), the simpler version – tiny YOLO v1

Introduction

Object detection from images are used treated as classification program in computer vision approaches. To detect objects, the objects' features are defined and stored in a vector data structure. The extract features used to be unique to the object, and a classifier is designed and trained by the feature vector as input and the labeled object as ground truth. The common classifier can be designed by using SVM or Decision Tree. The features are extracted by many methods, one of them is HOG – Histogram of Gradient. After the classifier is trained, the testing image will be feed, and a sliding window is used to walk through the entire image portion by portion. Each portion is checked by the classifier and for candidate detected object, it returns the coordinate and able to draw a box to cover the object in the image.

Sliding window + HOG + SVM is the method introduced in the class, and I have searched out relevant materials and found out that the object detection can be also solved as a regression method rather than classification. Here is a table conclusion about the two approaches:

classification	regression
Classification on portions of the image to determine objects, generate bounding boxes for regions that have positive classification results	Regression on the whole image to generate bounding boxes
1. sliding window + HOG 2. sliding window + CNN 3. region proposals + CNN	generate bounding box coordinates directly from CNN
RCNN, Fast-RCNN, Faster-RCNN	SSD, YOLO

In regression, one of a recent achievement that states faster processing speed and accuracy is called YOLO (you only look once). It reframes image pixels to bounding box coordinates and class probabilities with just convolutional networks and simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance.

In this project, we will implement the tiny YOLO v1 network, the reference paper can be found here
http://arxiv.org/abs/1506.02640

YOLO Output

YOLO cuts the input image to tunable size of SxS grid cell. If the center of the object is in a grid cell, the grid cell is used to detect the object. Each grid cell predicts B bounding boxes and the related confidence scores.

Confidence is defined as the Probability that the grid cell contains the object multiplied by Intersection over union of predicted bounding box over the ground truth:

$$\text{Confidence} = \Pr(\text{Object}) \times \text{IOU}_{\text{truth\_pred}}$$

So Confidence is zero if no object in the cell. Otherwise the best is when the object is in the cell, the Probability is one and Confidence is equal to IOU.

Each bounding box consists of 5 predictions:
1. x
2. y
3. w
4. h
5. confidence

The (x, y) coordinates is the center of the box relative to the bounds of the grid cell. The width and height are the predicted image size, and confidence is what just explained.

Each grid cell also predicts the conditional class probabilities, Pr (Class/Object) These probabilities are conditioned on the grid cell containing an object. It only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

At test time we multiply the conditional class probabilities and the individual box confidence predictions,

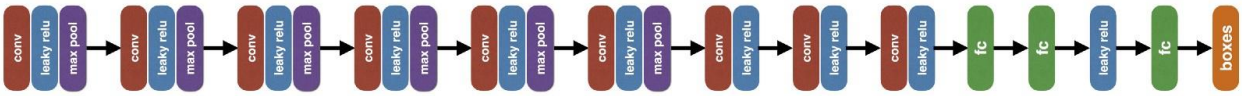
$$\Pr(\text{Class}|\text{Object}) \times \Pr(\text{Object}) \times \text{IOU}_{\text{truth\_pred}} = \Pr(\text{Class}) \times \text{IOU}_{\text{truth\_pred}}$$

which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.

The prediction is an output vector of each image. The size of the vector is calculated by parameters mentioned above, the size of the grid cell S by S, the number of bounding boxes, each box has 5 information (x, y, w, h, c) and the number of classifications (C). So at test time, the final output vector should have a length of $S \times S \times (B \times 5 + C)$

Neural Network Architecture

As suggested from the paper, the model architecture consists of 9 convolutional layers, followed by 3 fully connected layers. Each convolutional layer uses a Leaky RELU as activation function, with alpha of 0.1. The first 6 convolutional layers also have a 2x2 max pooling layers. The first 9 convolution layers can be understood as the feature extractor, whereas the last three fully-connected layers can be understood as the "regression head" that predicts the bounding boxes. There is a total of 45,089,374 parameters in the model.



Architecture of the tiny-YOLO-v1

The model network is implemented by using Keras api and it is in code cell 3 in the python notebook file, following the model summary.

The network need to be trained and I am using a pre-trained weight for this project. The weight is loaded into the model as in code cell 5. The pretrained weight data is got by the following training:

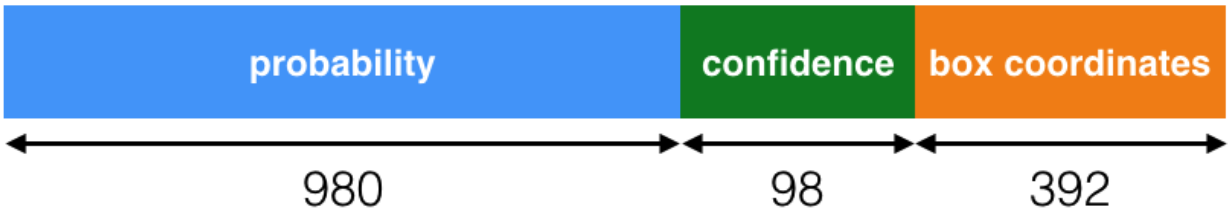
1: Training for classification
This model was trained on ImageNet 1000-class classification dataset. For this we take the first 6 convolutional layers followed by a followed by a fully connected layer.

2: Training for detection
The model is then converted for detection. This is done by adding 3 convolutional layers and 3 fully connected layers. The modified model is then trained on PASCAL VOC detection dataset.

The pre-trained weights file for this model are available at
https://drive.google.com/file/d/0B1tW_VtY7onibmdQWE1zVERxcjQ/view?usp=sharing

The model was trained on PASCAL VOC dataset. The parameters setting is S = 7, B = 2. PASCAL VOC has 20 labelled classes so C = 20. So the final output of the prediction, the length of the vector is:

output vector length = S x S x (B x 5 + C)
output vector length = 7 x 7 x (2x5 + 20)
output vector length = 1470.



The structure of the 1470 length vector is:

1. First 980 values corresponds to probabilities for each of the 20 classes for each grid cell. These probabilities are conditioned on objects being present in each grid cell.
2. The next 98 values are confidence scores for 2 bounding boxes predicted by each grid cells.
3. The next 392 values are co-ordinates (x, y, w, h) for 2 bounding boxes per grid cell.

In postprocessing, to generate the boxes from the 1470 vector information, it suggests using a probability higher than a certain threshold.

Also, a pre-processing to the input images is considered. This is because for this vehicle detection project, we can assume the vehicles are always appears in certain area in the image, especial the second half of the image, so the

first half of the image data (sky) is useless to the network. Certain preprocessing steps are covered such as cropping and resizing. Normalization is also a good one to normalize the image pixels to have value between -1 and 1. The preprocessing is done in code cell 6.

In code cell 7 is to create an image process pipeline for commonly usage. The strategy on using the predicted data is implemented in the help function from utils.py -- yolo_output_to_car_boxes() and draw_boxes(). It rejects output from grid cells below a certain threshold (0.2) of class scores, computed at test time. If multiple bounding boxes, for each class overlap and have an IOU of more than 0.4 (intersecting area is 40% of union area of boxes), then we keep the box with the highest-class score and reject the other box. The predictions (x, y) for each bounding box are relative to the bounds of the grid cell and (w, h) are relative to the whole image. To compute the final bounding box coordinates we must multiply `w` & `h` with the width & height of the portion of the image used as input for the network.

Testing

The first test in code cell 8 is a signal image test with the pipeline and result is:

The two vehicles are detected, and red boxes are drawing to cover them.

Then the test in code cell 9 ran through all the test images and gave:

The Video test in code cell 10, the output video file can be found in the repo named as “project_video_output.mp4”

Discussion

Yolo is easier to implement than Sliding window + HOG + SVM, and Yolo runs very fast. The video process only took about 4min with a GTX1060 GPU. But I not sure how is the framerate in real time running, and of course in real time the SOC board may not have such powerful processor. As the paper states YOLO can reach at 200fps but with very powerful desktop hardware. This YOLO network was pretrained with the dataset that has 20 classes for classification. If we only want to detect vehicles on the road, it is better to retrain the model from scratch, we can reduce the number of class, the parameter C to 1, only for vehicles. However, as for self-driving cars, it is still necessary that the car can detect different objects such as cars, trucks, animals, human, traffic light, etc.

Reference

1. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, You Only Look Once: Unified, Real-Time Object Detection, arXiv:1506.02640 (2015).
2. J. Redmon and A. Farhadi, YOLO9000: Better, Faster, Stronger, arXiv:1612.08242 (2016).
3. darkflow, https://github.com/thtrieu/darkflow
4. Darknet.keras, https://github.com/sunshineatnoon/Darknet.keras/
5. YAD2K, https://github.com/allanzelener/YAD2K