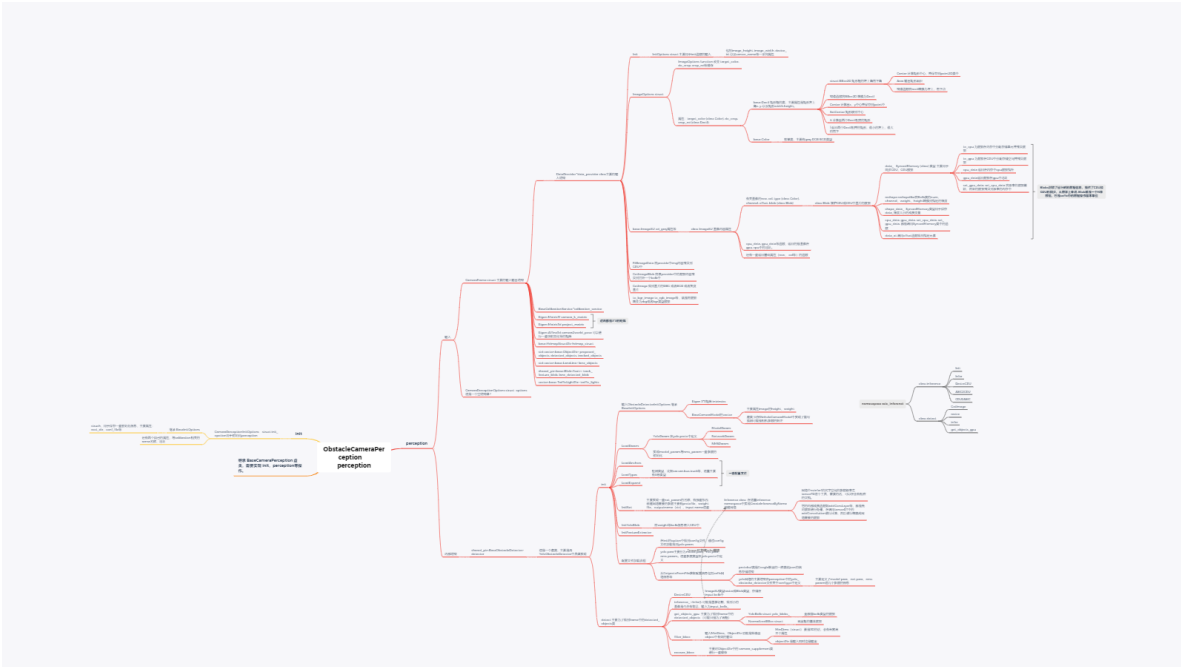
程序介绍

程序结构

我主要是阅读了camera perception相关的代码，其中包括了很多caffe相关、tensorRT相关、cuda c相关、c++线性代数库相关的知识，如需要可以找相应的材料进行学习。



Blob

其直观的可以把它看成一个有4纬的结构体（包含数据和梯度），而实际上，它们只是一维的指针而已，其4维结构通过shape属性得以计算出来（根据C语言的数据顺序）。

其成员变量有

```
protected:
  shared_ptr<SyncedMemory> data_;// 存放数据
  shared_ptr<SyncedMemory> diff_;//存放梯度
  vector<int> shape_; //存放形状
  int count_; //数据个数

```

常见的成员函数有

```
const Dtype* cpu_data() const; //cpu使用的数据
  void set_cpu_data(Dtype* data);//用数据块的值来blob里面的data。
  const Dtype* gpu_data() const;//返回不可更改的指针，下同
  const Dtype* cpu_diff() const;
  const Dtype* gpu_diff() const;
  Dtype* mutable_cpu_data();//返回可更改的指针，下同
  Dtype* mutable_gpu_data();
  Dtype* mutable_cpu_diff();
  Dtype* mutable_gpu_diff();
```

带mutable开头的意味着可以对返回的指针内容进行更改，而不带mutable开头的返回const 指针，不能对其指针的内容进行修改，

- 其中封装了syncedmemory类，该类主要是进行了cpu跟GPU数据类型的同步
- 第一版的代码我延续时使用了Blob数据结构，主要为了兼容其他的算法。

对于具体的图像而言blob是个四维数组(num，channels，height，width)，计算count=num×channels×height×width。
num是batch_size，channels是颜色通道数，RGB就是3;h和w分别就是图像的高和宽了。

对于全连接网络来说，blob是二维数组(num，datum)。

如果是卷积层，blob是四维数组，与图像数据相同;
如果是全连接层那么就是二维数组(num_inputs,num_outputs)。

Image8U

是对Blob类的一层封装，同时特定为image类型，不会有全连接类型，其中主要的属性有

```
  int rows_;  //图片row
  int cols_;  //图片col
  Color type_;  //rbg bgr gray三种
  int channels_;
  int width_step_;
  std::shared_ptr<Blob<uint8_t>> blob_;  //主要数据还是存储在blob中
```

分装了两种指针，Image8UPtr可变指针，Image8UConstPtr不可变指针。

除了blob中的xxx_data()属性之外，还有prt属性，可以指定一定的偏移。

配置信息流

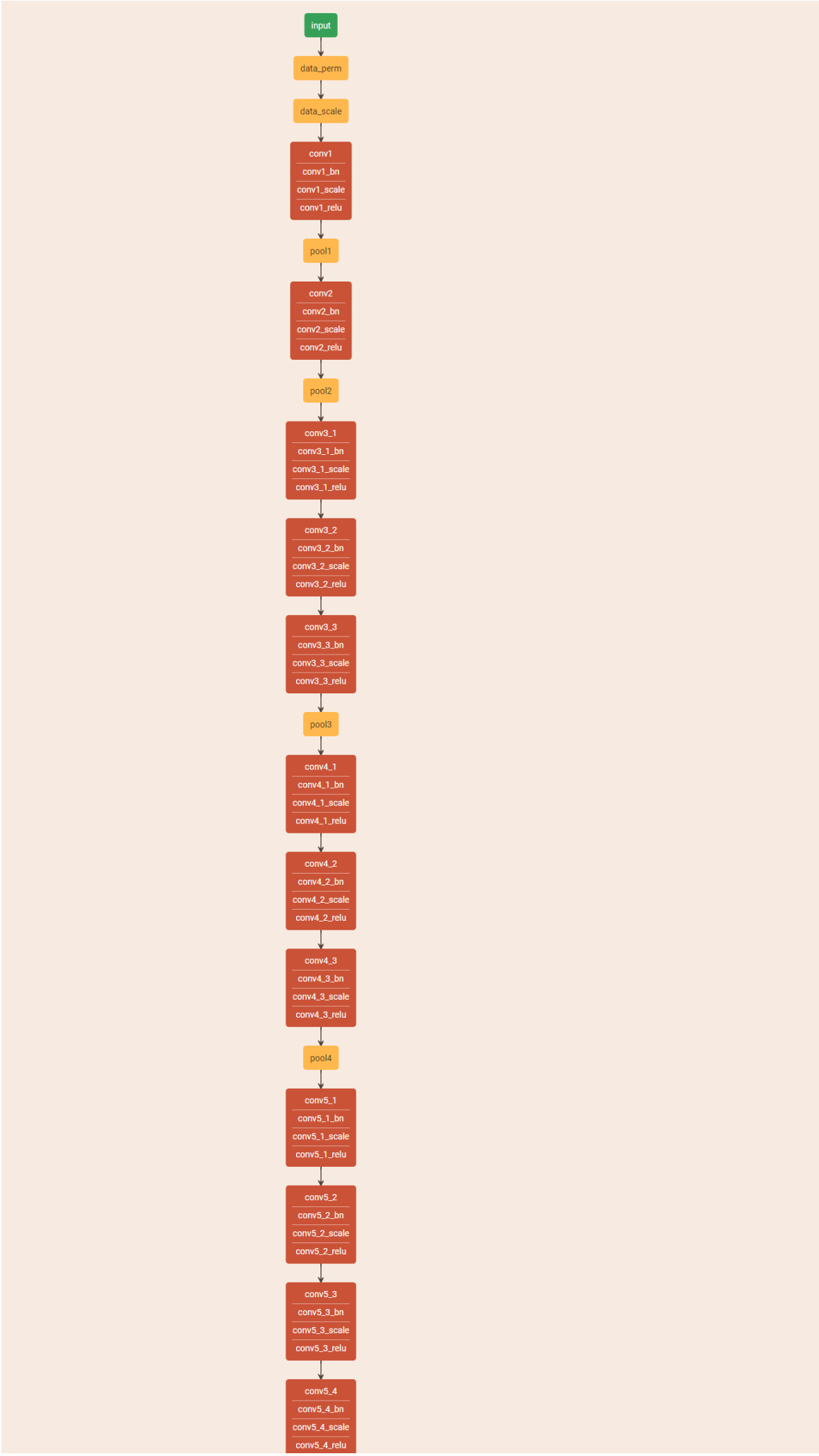
- 首先全局的配置信息在"/apollo/modules/perception/testdata/camera/app/conf/perception/camera/obstacle/obstacle.pt";中
- YoloObstacleDetector的配置信息在/apollo/modules/perception/production/data/perception/camera/models/yolo_obstacle_detector/conf.pt中，会分为model_param、net_param、nms_param三个主要的message，定义在modules\perception\camera\lib\obstacle\detector\yolo\proto\yolo.proto中。
- 在init函数中，通过GetProtoFromFile从config file中加载具体的配置参数
- 调用initnet函数对输入输出接口进行初始化，主要初始化了inference类跟其的input与output，之后在InitYoloBlob中进行了blob的初始化。

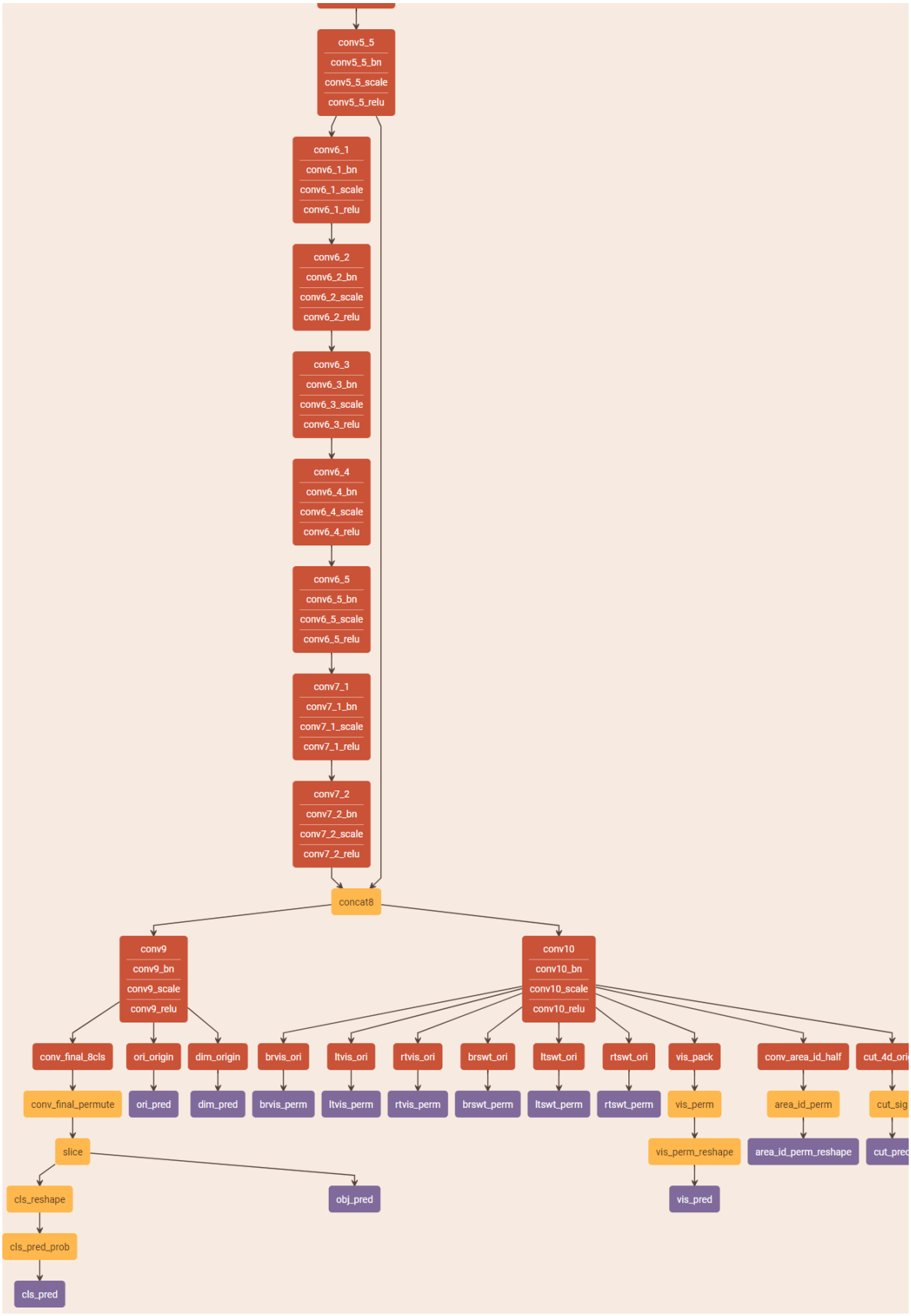
网络input output

使用网站查看caffe网络结构

apollo\modules\perception\production\data\perception\camera\models\yolo_obstacle_detector\3d-r4-half-config.pt

3d-r4-half-config


input
data_perm
data_scale
conv1
conv1_bn
conv1_scale
conv1_relu
pool1
conv2
conv2_bn
conv2_scale
conv2_relu
pool2
conv3_1
conv3_1_bn
conv3_1_scale
conv3_1_relu
conv3_2
conv3_2_bn
conv3_2_scale
conv3_2_relu
conv3_3
conv3_3_bn
conv3_3_scale
conv3_3_relu
pool3
conv4_1
conv4_1_bn
conv4_1_scale
conv4_1_relu
conv4_2
conv4_2_bn
conv4_2_scale
conv4_2_relu
conv4_3
conv4_3_bn
conv4_3_scale
conv4_3_relu
pool4
conv5_1
conv5_1_bn
conv5_1_scale
conv5_1_relu
conv5_2
conv5_2_bn
conv5_2_scale
conv5_2_relu
conv5_3
conv5_3_bn
conv5_3_scale
conv5_3_relu
conv5_4
conv5_4_bn
conv5_4_scale
conv5_4_relu



input

name	shape	describe
input	1 x 800 x 1440 x 3	image

output

name	shape
det1_loc_blob: "loc_pred"	50 x 90 x 64
det1_obj_blob: "obj_pred"	50 x 90 x 16
det1_cls_blob: "cls_pred"	4500 x 128
det1_ori_blob: "ori_pred"	50 x 90 x 32
det1_dim_blob: "dim_pred"	50 x 90 x 48
det2_loc_blob: "detect2_loc_pred"	-
det2_obj_blob: "detect2_obj_pred"	-
det2_cls_blob: "detect2_cls_pred"	-
det2_ori_conf_blob: "detect2_ori_conf_pred"	-
det2_ori_blob: "detect2_ori_pred"	-
det2_dim_blob: "detect2_dim_pred"	-
det3_loc_blob: "detect3_loc_pred"	-
det3_obj_blob: "detect3_obj_pred"	-
det3_cls_blob: "detect3_cls_pred"	-
det3_ori_conf_blob: "detect3_ori_conf_pred"	-
det3_ori_blob: "detect3_ori_pred"	-
det3_dim_blob: "detect3_dim_pred"	-
lof_blob: "lof_pred"	-
lor_blob: "lor_pred"	-
brvis_blob:"brvis_pred"	50 x 90 x 16
brswt_blob:"brswt_pred"	50 x 90 x 16
ltvis_blob:"ltvis_pred"	50 x 90 x 16
ltswt_blob:"ltswt_pred"	50 x 90 x 16
rtvis_blob:"rtvis_pred"	50 x 90 x 16
rtswt_blob:"rtvis_pred"	50 x 90 x 16
feat_blob:"conv_feat"	-
area_id_blob:"area_id_pred"	72000 x 4
visible_ratio_blob:"vis_pred"	72000 x 4
cut_off_ratio_blob:"cut_pred"	50 x 90 x 64

yolo_param

name	shape
input	1 x 800 x 1440 x 3
data_perm	1 x 3 x 800 x 1440
conv1	16 x 800 x 1440 (batch 维度接下来省略)
pool1	16 x 400 x 720
conv2	32 x 400 x 720
pool2	32 x 200 x 360
conv3_1	64 x 200 x 360
conv3_2	32 x 200 x 360
conv3_3	64 x 200 x 360
pool3	64 x 100 x 180
conv4_1	128 x 100 x 180
conv4_2	64 x 100 x 180
conv4_3	128 x 100 x 180
pool4	128 x 50 x 90
conv5_1	256 x 50 x 90
conv5_2	128 x 50 x 90
conv5_3	256 x 50 x 90
conv5_4	128 x 50 x 90
conv5_5	256 x 50 x 90
conv6_1	512 x 50 x 90
conv6_2	256 x 50 x 90
conv6_3	512 x 50 x 90
conv6_4	256 x 50 x 90
conv6_5	512 x 50 x 90
conv7_1	512 x 50 x 90
conv7_2	256 x 50 x 90
concat8	**(256 + 256) x 50 x 90**
conv9	512 x 50 x 90
conv10	512 x 50 x 90
conv_final_8cls	208 x 50 x 90

name	shape
conv_final_permute	50 x 90 x 208
slice	loc_pred(50 x 90 x 64), obj_perm(50 x 90 x 16), cls_perm(50 x 90 x 128)
cls_reshape	72000 x 8
cls_pred_prob	72000 x 8
cls_pred	**4500 x 128**
obj_pred	**50 x 90 x 16**
loc_pred	**50 x 90 x 64**
dim_origin	48 x 50 x 90
dim_pred	**50 x 90 x 48**
ori_origin	32 x 50 x 90
ori_pred	**50 x 90 x 32**
brvis_ori	16 x 50 x 90
brvis_perm	**50 x 90 x 16**
ltvis_ori	16 x 50 x 90
ltvis_perm	**50 x 90 x 16**
rtvis_ori	16 x 50 x 90
rtvis_perm	**50 x 90 x 16**
brswt_ori	16 x 50 x 90
brswt_perm	**50 x 90 x 16**
ltswt_ori	16 x 50 x 90
ltswt_perm	**50 x 90 x 16**
rtswt_ori	16 x 50 x 90
rtswt_perm	**50 x 90 x 16**
vis_pack	64 x 50 x 90
vis_perm	50 x 90 x 64
vis_perm_reshape	72000 x 4
vis_pred	**72000 x 4**
conv_area_id_half	64 x 50 x 90
area_id_perm	50 x 90 x 64
area_id_perm_reshape	**72000 x 4**

name	shape
cut_4d_origin	64 x 50 x 90
cut_sig	64 x 50 x 90
cut_pred	**50 x 90 x 64**

Detect class

```
enum class ObjectSubType {
  UNKNOWN = 0,
  UNKNOWN_MOVABLE = 1,
  UNKNOWN_UNMOVABLE = 2,
  CAR = 3,
  VAN = 4,
  TRUCK = 5,
  BUS = 6,
  CYCLIST = 7,
  MOTORCYCLIST = 8,
  TRICYCLIST = 9,
  PEDESTRIAN = 10,
  TRAFFICCONE = 11,
  MAX_OBJECT_TYPE = 12,
};
```

Detect

- ResizeGPU

  在进行resize之前需要将camerafarmer中的额data_provider中的image放到image中，image_是一个image8U类型的数据，resize的结果放到input_blob中，这是在inference中已经申请了空间的数据结构，用于infer的输入。

- Infer

  通过之前配置好的网络以及input_blob中的数据，进行inference

  ```
    for (auto name : output_names_) {
      auto blob = get_blob(name);
      if (blob != nullptr) {
        blob->mutable_gpu_data();
      }
  ```

  可以看到的是，他将得到网络所有output的结果，保存在最后保存在yolo_blob_中，感觉这个inference更多的是为后面的get_object_gpu（Yolo）服务的

- get_objects_gpu
  - 实现在modules\perception\camera\lib\obstacle\detector\yolo\region_output.cu中。
  - 其中主要是对infer的output进行处理，输入参数说明如下所示

```
yolo_blobs_; //inference得到的结果
stream_; //cuda所用到的流
types_; //type类型，主要有car，per等
nms_; //NMSParam
yolo_param_.model_param(); //yolo的module param
light_vis_conf_threshold_; //float类型的数据，阈值
light_swt_conf_threshold_; //float类型的数据，阈值
overlapped_.get();  //bool类型的blob
idx_sm_.get(); //int类型的blob
&(frame->detected_objects); //传入的引用，最终改函数的结果保存在这。
```

- 开始是进行一些准备信息的工作，提取出yolo_blob_中的信息，判断lof、lor等信息，其中还有一些具体的维度计算没有跟踪。
- get_object_kernel函数，猜测可能是检测用的网络用的函数，没太看明白中间在干些啥。
- 对于每一种物体类别，调用apply_nms_gpu()函数去除多余的检测框。两个apply_nms_gpu，第一个在for之外，其实是对最大的confidence的使用NMS算法，在for之中的是分别对剩下的使用NMS算法，最终的结果保存在indices中。
- 最后对剩下的框，填充物体的种类跟score，最后的fill box相关就是将obj->camera_supplement.box.xmin这些相关的信息填入cameraframe中。

对于每一种物体类别，调用apply_nms_gpu()函数去除多余的检测框。这个函数首先先用阀值过滤下，把confidence小于0.8的干掉，剩下的框的index和confidence放在idx和confidence两个vector中。然后把剩下的元素按confidence排个序。接着调用compute_overlapped_by_idx_gpu()函数计算这些框间的重叠关系。当两个框间的IoU（即Jaccard overlap）大于0.4时，算两者重叠。基于这个结果，就可以调用apply_nms()执行NMS算法。结果放在indeces这个成员中。indices是类别到物体框index数组的映射。

- filter_bbox

  这就是将不合理的框筛选出去，如左上角的点的坐标大于右上坐标的点。

- feature_extractor_->Extract

  根据初始化中使用的是"TrackingFeatureExtractor"，所以调用的是modules\perception\camera\lib\feature_extractor\tfe\tracking_feat_extractor.cc中的extract。

  这里是使用输入的box的位置信息，得到该box表示的是什么样的东西（如person等），主要注意的是这里也有一个网络。

```
float *rois_data =
    feature_extractor_layer_ptr->rois_blob->mutable_cpu_data();
for (const auto &obj : frame->detected_objects) {
    rois_data[0] = 0;
    rois_data[1] =
        obj->camera_supplement.box.xmin * static_cast<float>
(feat_width_);
    rois_data[2] =
        obj->camera_supplement.box.ymin * static_cast<float>
(feat_height_);
    rois_data[3] =
        obj->camera_supplement.box.xmax * static_cast<float>
(feat_width_);
    rois_data[4] =
```

```
        obj->camera_supplement.box.ymax * static_cast<float>
(feat_height_);
    ADEBUG << rois_data[0] << " " << rois_data[1] << " " <<
rois_data[2]
        << " " << rois_data[3] << " " << rois_data[4];
    rois_data += feature_extractor_layer_ptr->rois_blob->offset(1);
}
feature_extractor_layer_ptr->pooling_layer->ForwardGPU(
    {feat_blob_, feature_extractor_layer_ptr->rois_blob},
    {frame->track_feature_blob});
```

- recover_bbox

  这个函数主要做的是一些与坐标转换相关的处理，upper_left/uppper_right中填的是相对于ROI（图片下面1920 x 768）中并且归一化到[0，1]范围内的。比如xmin/ymin/xmax/ymax为(0.552336, 0.27967, 0.583794, 0.344488)。这个坐标会转换到ROI中的像素坐标。转换后x/y/w/h为(1060, 526, 60, 49)。之后还会和图像的大小做一个并，把那些超出图像的部分切掉。最后如果框的边界是在图像的边上的，会设置VisualObject的trunc_width/trunc_height成员。

最终可以看到detect中各项函数使用的时间如下所示。

name	time(ms)	describe
start	0.003	get input blob
GetImageBlob	0.014	from provider to input blob
ResizeGPU	0.014	resize
Infer	**7.367**	**inference**
get_objects_gpu	**4.184**	**get objects**
filter_bbox	0.042	filter not create box
Extract	**0.115**	**Extract features**
recover_bbox	0.014	

可以通过perception的tool的offline中的offline_obstacle_pipeline.cc进行离线的算法测试，最终生成测试图片如下所示。

最终的object检测的结果其实会保存在输入的frame.detected_objects.camera_supplement中，其中画出的box主要是在box中，这里的主要是2D的矩形，许在真实的系统中其实还进行了3D重建。

name	describe
xmin, ymin	矩形的左上角点
xmax, ymax	矩形的右上角的点

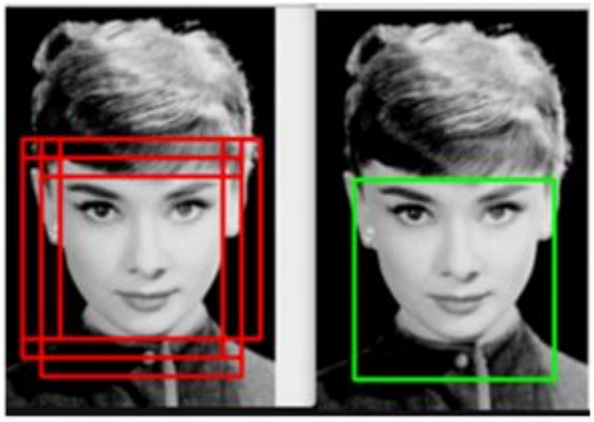
给出一该图片的检测输出

name	value
car	547 670 800 857 0.949245
car	1213 633 1789 988 0.934825
car	185 553 636 1024 0.89498
car	851 682 929 741 0.88613
car	905 685 948 727 0.605641
car	1010 685 1034 710 0.528782
pedestrian	1751 573 1852 875 0.951115

NMS算法

根据候选框的类别分类概率做排序: A < B < C < D < E < F

1. 先标记最大概率F
2. 将F与A-E进行IOU，设定阈值，要是小的舍掉，假设B，D舍掉
3. 在A C E中选取最大的E重复上述步骤



工具介绍

tensorRT

TensorRT是NVIDIA推出的一个高性能的深度学习推理框架，可以让深度学习模型在NVIDIA GPU上实现低延迟，高吞吐量的部署。TensorRT支持Caffe，TensorFlow，Mxnet，Pytorch等主流深度学习框架。TensorRT是一个C++库，并且提供了C++ API和Python API，主要在NVIDIA GPU进行高性能的推理(Inference)加速。

程序在运行的过程中会出现INT8 mode not support的字样，这是因为2070super不支持一个叫做`DP4A`的指令集优化，换上支持此指令集的GPU如2080TI会得到3-5倍的加速

tensorRT的相关知识点比较多，我主要参看了这篇博客，如果需要将默认的yolo网络修改为自己想要的网络，需要对tensorRT有一定的认识

nvinfer1

这是tensorRT中的一个namespace，他比较重要的地方是实现了网络中的不同层的GPU加速，在apollo中主要的优化就是争对apollo这个系统所特有的数据格式进行了数据处理，计算函数还是使用的tensorRT自带的计算函数，具体函数的使用可以查看官方文档，文档中给了具体函数的使用接口，以及功能介绍。

caffe

相对于工程性的项目，caffe的使用感觉是比tensorflow这些会方便一点，特别时融合了很多其他框架的时候，caffe的优势就体现出来了，caffe完全以配置信息的形势呈现，修改对应的网络模型只需要修改对应的配置文件，不需要重新对代码进行修改或者编译，我学习过程主要参考了这个人的博客，以及对源码进行了一定额阅读，其实在apollo的算法模块很多都是caffe的源码，需要的话可以进行阅读。

gtest

google开源测试工具，可以很方便的争对你开发的每一个函数进行快速的测试，具体可以查看官方教程

protobuf

google开源的类似json或者xml，是一种轻便高效的结构化数据存储格式，可以用于结构化数据串行化，或者说序列化。具体可查看官方教程，以及一些不错的博客。

cuda c

为了调用GPU进行运算，需要用到cuda c给出的API，Apollo也是使用cuda c这样的框架进行的异构设备的管理。

eigen

Eigen 是C++语言里的一个开源模版库，支持线性代数运算，矩阵和矢量运算，数值分析及其相关的算法。在Apollo中需要用这个库中的矩阵操作，包括一些矩阵乘法，放射变换等。在cameraframe类中有挺多。

opencv

会使用到cv中一些图像处理的函数，比如resize中的线性插值等。