RetinaNet及Yolov3的部署过程

1、使用TensorRT API部署需要掌握哪些知识?

- 掌握算法的网络结构、预处理实现、后处理实现
- 部署算法所需要的TensorRT api
- TensorRT plugin相关接口的作用
- cuda基础知识

2、如何快速部署一个算法?

拿基于锚框检测的Yolov3举例

对于一个之前没有了解过的算法，如何快速进行部署呢？阅读该算法的论文？显然，阅读论文对于部署这件事没有太大帮助（当然没有否认阅读论文本身的意义，要对算法有深刻的认识，要用好算法，离不开阅读论文）。要理解"没有太大帮助"这句话，首先需要明确以下几个问题的答案。

- 需要了解算法是怎么训练的?
- 需要了解算法是怎么采集正负样本的?
- 需要了解其损失函数怎么运作的?
- 到底需要了解算法的什么东西?

以上前三点均不需要，因为预测过程仅仅是**前向推理**，因此我们不需要关注其训练细节、正负样本的采样方法、损失函数的运作原理。这很大程度上减少了部署所需的先验知识。

我们只需要了解推断过程是怎样的：

- 数据预处理
- 预处理的数据经过backbone、neck，得到特征数据
- 将特征数据经过head，得到锚框的**坐标偏移值**、**置信度**、**类别**
- 在特征图上铺设锚框，得到一系列锚框坐标。
- 后处理：
    - 将**锚框坐标**与**坐标偏移值**进行解码运算，得到预测框的坐标
    - 利用nms过滤掉某一类别物体的冗余检测框
- 将后处理后的检测框绘制在输入图片上

2.1 预处理的部署

在部署算法的过程中，发现推断时的预处理大多包含以下两点（与训练时不同，训练时为了增加模型的鲁棒性，会进行一些图像变换）：

- resize
- 图像归一化

部署resize时遇到的坑：

不同库中的resize处理方式不同，且知道他的resize方式后，也难以用cuda进行加速。

例如：大部分框架中会使用Pillow和Opencv中的resize进行缩放，但是PIL是一个python库，并没有对应的c++、cuda实现，当其使用该方式进行缩放时，便无法使用cuda进行加速。而当框架中使用Opencv中的resize进行缩放时，同样也难以进行cuda加速。

也许你会有以下疑问：

- 为什么opencv中的resize也难以进行cuda加速呢？
- opencv的resize不是有对应的cuda版本吗？直接调用cuda版本对应的api不就行了？

首先，python版本的opencv和c++版本的opencv结果是一致的，且opencv c++版本是开源的。理论上，掌握其源码实现，并用cuda实现就能解决问题。但是opencv库中的resize源码非常繁琐，写法复杂，代码量大，且里面用到一些优化算法，难以读懂，因此使用cuda复现该算法难度过大，不具可行性。

其次，c++版本的opencv::resize和cv::cuda::resize版本的结果不同，一旦两者结果不同，其推断结果便会不一致。

如何解决这个问题？首先在github上找到一个cuda版本实现的resize，测试其resize的效果（对于不同尺寸，是否resize后均能正常显示图像，因为有的实现版本不理想，在resize后，图像存在重影等问题）。在测试其效果后，将其resize使用pycuda实现，并替换掉框架中的resize函数，此时，框架下的resize和部署时的resize结果便能够保持结果一致。

图像归一化的部署：

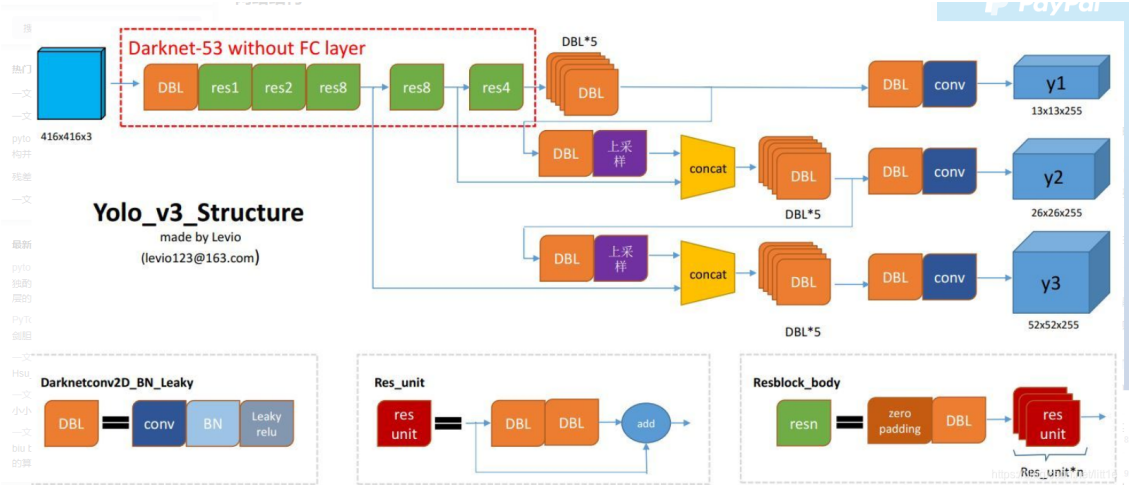
该部分较为简单，在确定其归一化公式及参数，使用cuda实现即可。

2.2 backbone、neck、head的部署

使用TensorRT api要对网络细节比较了解，这是毋庸置疑的。但是实现起来是否真的非常困难？答案是否定的。

那么如何能够快速部署网络结构呢？按照以下步骤便能顺利将网络搭建起来：

- **网络结构初探：通过博客，找到该算法的网络结构图**



首先看到网络结构图后，将结构图分解成三个模块：backbone、neck、head。一些开源框架都是按照此方式对网络结构进行拆分。

  - backbone：通过卷积进行特征提取。
  - neck：浅层、深层特征进行融合，以达到特征增强的目的
  - head：基于锚框算法的head，其输出表示锚框的偏移量、置信度、类别分数信息。

  这三个结构本质上就是“卷积”、“激活”运算（yolov3中无池化操作）。

先通过该网络图，大概了解了这三个模块是由哪些小模块组成，数据是怎么流通的。此时不必知道每一个小模块具体由几个卷积组成、里面有几个激活函数。在有个大概印象之后，通过源码了解其实现细节。

- **网络结构再探：调试源码，熟悉具体细节**

  一般开源框架下的算法会提供一个预训练模型，使用该预训练模型能够将模型跑通，直接拿预训练模型进行推断调试，了解其网络结构实现。

  怎么从这么多的源码中跳脱出来的同时，掌握网络结构的细节呢？

1. 先看下源码中最外层函数做了哪些事情

```
x = self.extract_feat(img)
outs = self.bbox_head(x)
```

此处便是mmdetection中对于单阶段算法的特征提取、head预测最外层实现。

2. 再看self.extract_feat(img)做了哪些事情

```
def extract_feat(self, img):
    """Directly extract features from the backbone+neck."""
    x = self.backbone(img)
    if self.with_neck:
    x = self.neck(x)
    return x
```

可以很清楚的看到，分为两步：骨干网特征提取、neck特征融合。

3、在调试模式下，看每一个部分的网络结构

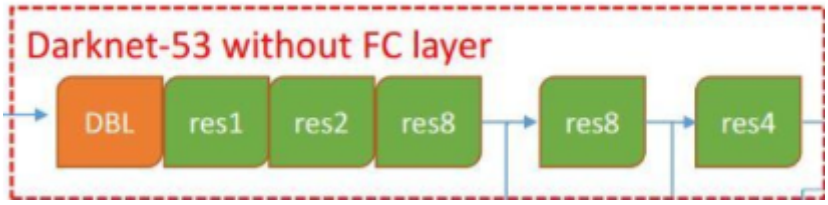
拿backbone举例，yolov3的backbone为darknet53。

```
YOLOV3(
  (backbone): Darknet(
    (conv1): ConvModule(
        DBL模块
    )
    (conv_res_block1): Sequential(
      (conv): ConvModule(
         DBL模块
      )
      (res0): ResBlock(
          DBL模块
          DBL模块
      )
    )
    (conv_res_block2): Sequential(
    )
    (conv_res_block3): Sequential(
    )
    (conv_res_block4): Sequential(
    )
    (conv_res_block5): Sequential(
    )
  )
```

而DBL模块如下所示：

```
(conv): Conv2d()
(bn): BatchNorm2d()
(activate): LeakyReLU()
```

再进行简化，和backbone的结构图对比以下：




```
YOLOV3(
  (backbone): Darknet(
    (conv1): ConvModule(
    )
    (conv_res_block1): Sequential(
    )
    (conv_res_block2): Sequential(
    )
    (conv_res_block3): Sequential(
    )
    (conv_res_block4): Sequential(
    )
    (conv_res_block5): Sequential(
    )
  )
```

对应的TensorRT实现如下：

```
void EngineDetYolov3::constructDarknet(INetworkDefinition * network, ITensor &
input, std::vector<ILayer*>& darknetOut)
{
    enum NetworkPart networkPart = BACKBONE;
    auto conv1 = convBnLeaky(network, input, 32, 3, 1, 1, "conv1", networkPart);

    //搭建darknet的5个conv_res_block模块
    auto crb1 = convResBlock1(network, *conv1->getOutput(0));
    auto crb2 = convResBlock2(network, *crb1->getOutput(0));
    auto crb3 = convResBlock3(network, *crb2->getOutput(0));
    auto crb4 = convResBlock4(network, *crb3->getOutput(0));
    auto crb5 = convResBlock5(network, *crb4->getOutput(0));

    darknetOut.push_back(crb3);
    darknetOut.push_back(crb4);
    darknetOut.push_back(crb5);
}
```

其中的每个函数此处不进行具体展开。

2.3 部署算法所需要的TensorRT api

从上面可以看到，部署该算法，主要是不断调用DBL模块，进行搭建网络，而此处主要进行了三个操作：卷积、归一化、激活。

对应于TensorRT上的api，如下所示。

```
//卷积
IConvolutionLayer *conv = network->addConvolutionNd(input, outch, DimsHW(ksize, ksize), mWeightMap[convName], emptywts);
//归一化
IScaleLayer* scale = network->addScale(input, ScaleMode::kCHANNEL, shift, scale, power);
//Leaky relu激活
IActivationLayer* relu = network->addActivation(*bn->getOutput(0), ActivationType::kLEAKY_RELU);
```

2.4 TensorRT plugin的工作原理

为什么需要plugin?

一个完整的算法可以分为两个部分：与网络结构有关的部分，与网络结构无关的部分。

- 与网络结构有关的部分：即backbone、neck、head三个部分，从本质上说，就是**带有权重的操作。**
- 与网络结构无关的部分：**不包含训练权重的操作。**

与网络结构有关的部分，其算法相对固定。与框架无关。例如，一个卷积操作，其在pytorch下、tensorflow下、mmdetection中、detectron2中可能表现形式不同，但是其本质都是一个算法，此处相对稳定。由于其具有稳定的性质，因此，TensorRT开发人员可以开发出一个统一的api，使用者在调用api时，只需给出训练好的权重，以及输入数据，便能够得到算法的输出结果，而该结果与框架无关。

与网络结构无关的部分，其与框架有关，同一算法在不同框架下可能实现细节不同。例如，后处理，在不同框架下后处理流程可能不同，因此TensorRT开发人员不可能开发出一个通用的后处理算法。既然TensorRT无法对该类算法预先实现，那么TensorRT便将此实现全权交给使用者了，其只给出了实现规范。该实现规范即为Plugin类，用户按照规范实现其各种api，使得TensorRT能够以一种统一的方式，对用户的需求进行实现。

那么一个Plugin类该有哪些接口?

- 算法有输入和输出数据，输入数据由前一层网络的输出给出，这个是确定的。TensorRT需要知道该算法的输出信息，例如输出的个数、输出数据的维度，以便TensorRT分配资源。因此有以下两个api。

  ```
  //获取输出节点数量
  virtual int32_t getNbOutputs () const =0
  //获取输出节点的维度
  virtual Dims getOutputDimensions (int32_t index, const Dims *inputs, int32_t nbInputDims)=0
  ```

- TensorRT需要提供一个接口给用户实现算法。

  ```
  virtual int32_t enqueue (int32_t batchSize, const void *const *inputs, void **outputs, void *workspace, cudaStream_t stream)=0
  ```

- 用户在使用cuda实现算法的时候，需要用到临时的显存，存放临时变量，因此TensorRT需要知道所需显存的大小，以便分配资源。

  ```
  virtual size_t  getWorkspaceSize (int32_t maxBatchSize) const =0
  ```

- 在模型转换的时候，需要将该算法需要用到的参数保存到本地文件，使得TensorRT在推断阶段，能够反序列化，得到这些参数，并将这些参数提供给算法，用于算法的运算。

```
//plugin的构造函数，在搭建网络时，调用该构造函数
detectron2::Yolov3AnchorPlugin::Yolov3AnchorPlugin(const AnchorParamsYolov3*
paramsIn, int numLayer)
//序列化参数至本地
virtual void serialize (void *buffer)=0
//计算序列化时用到的参数所占用的内存大小
size_t detectron2::Yolov3AnchorPlugin::getSerializationSize() const
//plugin构造函数，在推断时进行调用，作用是：反序列化参数，并使用反序列化后的参数初始化该
plugin成员变量
detectron2::Yolov3AnchorPlugin::Yolov3AnchorPlugin(const void* data, size_t
length)
```

2.5 cuda基础知识

略。

2.6 一个后处理python语句的cuda实现**

python语句：sigmoid实现。

```
for i in range(self.num_levels):
    # get some key info for current scale
    pred_map = pred_maps_list[i]
    pred_map = pred_map.permute(0, 2, 3,
                                1).reshape(batch_size, -1,
                                           self.num_attrib)
    pred_map_conf = torch.sigmoid(pred_map[..., :2])
    ......
```

cuda实现

```
__global__ void sigmoid(const float * input, float* output, int dims)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid >= dims)
    {
        return;
    }
    output[tid] = exp(input[tid]) / (1 + exp(input[tid]));
}
```

3、小结

使用TensorRT api进行部署时，其不存在技术难点，部署主要难点在于实现plugin。例如，需要将后处理中的每一条语句都使用cuda实现，一条python语句用cuda实现较为简单，但该当python较多时，则容易出错，且cuda程序调试不方便。主要出错点：多线程下，输入，输出的索引对应关系容易出错。