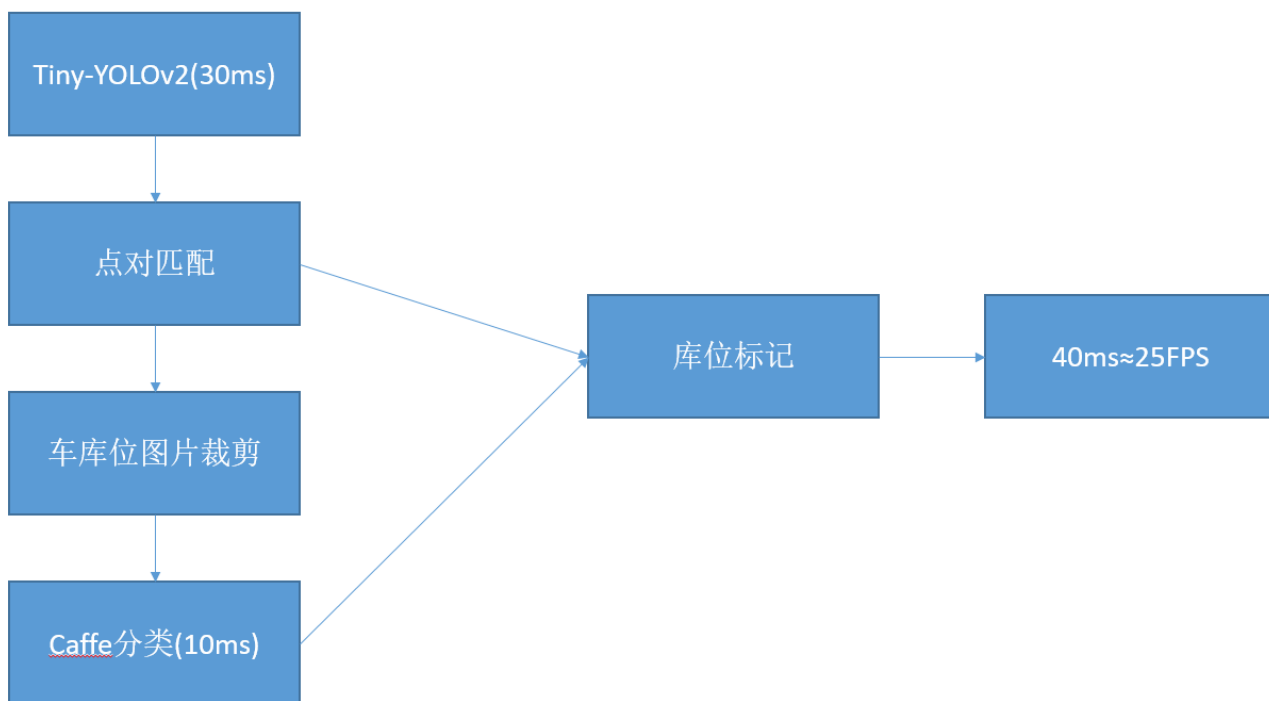


# NVIDIA JETSON TX2

## tensorRT 加速 Caffe 实战



### 1.准备工作

首先，请参照NVIDIA JETSON TX2刷机指南给TX2刷机，部署好全部所需开发环境：CUDA 8.0，cuDNN，tensorRT等。建议宿主机采用Jetpack 3.1为TX2刷机。

### 2.caffe配置

首先，配置所需要的依赖项：

```
sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libhdf5-serial-dev protobuf-compiler
sudo apt-get install -no-install-recommends libboost-all-dev
```

注意，这里这一步比caffe官方网站给出的教程少了一个libopencv-dev，经过验证不影响后续操作的正常进行，所以不要写。如果写了会报错，提示一些依赖项不满足。

然后继续：

```
sudo apt-get install -no-install-recommends libboost-all-dev
```

接下来是python相关的安装：

```
sudo apt-get install python-dev
sudo apt-get install python-numpy
sudo apt-get install python-sklearn
sudo apt-get install python-skimage
sudo apt-get install python-protobuf
```

如果还没有在TX2上安装pip（如果你是刚刚好的机器，应该就是没有的吧），需要安装pip：

```
sudo apt-get install python-pip
sudo pip install --upgrade pip
```

接下来安装Google的glog和gflags以及imdb依赖项：

```
sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev
```

接下来安装git，并且下载caffe的源码：

```
sudo apt-get install git
git clone https://github.com/BVLC/caffe.git
```

这一步可能由于网络的问题下载失败，多试几次。（笔者的下载网速只有几十K）

实在不行直接去github把zip文件download下来再拷贝到TX2上也可以。

下载完成后，进入源码的目录：

```
cd caffe-master
```

这一步文件夹名称可能不一样，可能是caffe-master，取决于你是怎么下载源码的，无关紧要。

文件夹下有**Makefile.config.example**文件，我们需要将其复制为**Makefile.config**文件：

```
cp Makefile.config.example Makefile.config
```

接下来需要修改**Makefile.config**和**Makefile**两个文件：

```
sudo gedit Makefile.config
```

如果你熟悉Vim，也可以使用Vim进行编辑：

```
sudo vim Makefile.config
```

在**Makefile.config**文件中，去掉**USE\_CUDNN := 1**前的注释，启用cuDNN

然后去掉**WITH\_PYTHON\_LAYER := 1**前面的注释

然后将文件中的

```
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include
```

改为:

```
INCLUDE_DIRS := $(PYTHON_INCLUDE) /usr/local/include /usr/include/hdf5/serial/
```

接下来修改**Makefile**文件, 将文件中的

```
LIBRARIES += glog gflags protobuf boost_system boost_filesystem m hdf5_hl hdf5
```

改为:

```
LIBRARIES += glog gflags protobuf boost_system boost_filesystem m hdf5_serial hl hdf5_serial
```

然后:

```
sudo make clean  
sudo make -j8
```

其实上面的指令我自己写的是**sudo make -j4**, jn的n好像代表处理器核心数量? 会同时n线程执行make, 我查到tx2好像是4核的, 就用了j4, 经过尝试可以执行, -j8没试过, 可自行尝试。

如果make过程报错, 提示缺少依赖, 请按照提示安装相关内容。例如, 可能出现缺少libatlas的依赖, 这是一个线性代数运算库, 请输入以下指令安装:

```
sudo apt-get install libatlas-dev
```

由于笔者记忆有限, 可能会有其他缺少依赖的报错, 不是大问题, 缺什么补什么即可。关键部分的配置上文都已给出。

下面假设你已经成功make了caffe的源码, 接下来为了调用pycaffe, 需要继续make:

```
make pycaffe
```

过程中可能报错, 按提示进行, 也是缺什么补什么。

如果没有报错, 或者有报错并且按照报错提示安装好所需依赖, 应该可以正常的make好pycaffe。

然而, 此时如果用python去import caffe, 仍然无法import成功。需要将caffe加入python的环境变量中:

```
sudo gedit /etc/profile
```

在打开的profile文件的最后一行添加如下内容：

```
export PYTHONPATH=/home/nvidia/caffe-master/python:$PYTHONPATH
```

中间的路径为你的caffe文件夹下的python文件夹，根据自己的情况调整。

修改后保存并关闭文件，执行以下指令让其生效：

```
source /etc/profile
```

执行这条指令后，命令行的nvidia@tegra-ubuntu:~\$会变成白色，说明成功了。不过这个方法每次用python之前都要执行**source /etc/profile**，应该有更好的解决方案，不过这里笔者未作查找。

接下来尝试用python导入caffe：

```
python
import caffe
```

如果上面的教程没有全部执行，遗漏了一些依赖，可能会出现报错，例如：

```
ImportError: No module named skimage.io
```

解决方案很简单，缺什么补什么：

```
pip install -U scikit-image
```

一般教程都会这样告诉你，然而如果你执行上面的指令来安装，速度会奇慢无比，还可能掉线，这里就需要更改pip源到国内镜像，可以显著提升下载速度：

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple -U scikit-image
```

这样你的下载速度就起飞了。以后需要pip install任何东西都可以在指令里加上-i <https://pypi.tuna.tsinghua.edu.cn/simple>。这样将源换成清华的镜像，笔者的速度甚至达到了每秒10m。

最后，重新尝试import caffe。

```
nvidia@tegra-ubuntu:~$ sudo gedit /etc/profile
[sudo] password for nvidia:

(gedit:26423): Gtk-WARNING **: Calling Inhibit failed: GDBus.Error:org.freedesktop.DBus.Error.ServiceUnknown: The name org.gnome.SessionManager was not provided by any .service files

** (gedit:26423): WARNING **: Set document metadata failed: Setting attribute metadata::gedit-position not supported
nvidia@tegra-ubuntu:~$ source /etc/profile
nvidia@tegra-ubuntu:~$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import caffe
>>> 
```

如果没报错，恭喜你在TX2上成功配置了caffe的开发环境。

### 3.pyCaffe

在使用tensorRT加速caffe之前，我们先直接使用pyCaffe进行inference，并统计其推断所需时间：

首先是导入所需模块：numpy，caffe和用来统计时间的time。

然后设置caffe采用GPU模式。这里也可以采用CPU模式，但是速度在**800ms**左右，采用GPU模式速度可以提升十倍以上，在**60ms**左右。

接下来设置好caffe模型的prototxt文件和caffemodel文件的路径，并且使用caffe.Net函数构建网络。

最后一行是导入均值文件，图片减去均值再训练，会提高训练速度和精度。同样的，在测试阶段减去均值图片也会提高预测精度。

```
import numpy as np
import caffe
import time

caffe.set_device(0)
caffe.set_mode_gpu()
model_def = './deploy.prototxt'
model_weights = './mycaffenet_train_iter_450000.caffemodel'

net = caffe.Net(model_def, model_weights, caffe.TEST)

mean = np.load('./mean.npy').mean(1).mean(1)
```

接下来是设置transformer，不知道这个怎么翻译了，就是用来调整输入数据结构的东西。

matplotlib加载的image是像素[0-1],图片的数据格式[width,height,channels]，RGB

caffe加载的图片需要的是[0-255]像素，数据格式[channels,width,height]，BGR，所以需要转换

```
transformer = caffe.io.Transformer({'data':net.blobs['data'].data.shape})
transformer.set_transpose('data', (2,0,1))
transformer.set_mean('data', mean)
transformer.set_raw_scale('data', 255);
transformer.set_channel_swap('data', (2,1,0))#swap channels from RGB to BGR

net.blobs['data'].reshape(1,3,227,227)
```

然后就是读取图片，transform，前向传播预测结果以及输出了。

```
img_path = input('img path:')

#start = datetime.datetime.now()
start = time.clock()
image = caffe.io.load_image(img_path)
transformed_image = transformer.preprocess('data', image)
net.blobs['data'].data[...] = transformed_image
output = net.forward()
output_prob = output['prob'][0]
#end = datetime.datetime.now()
end = time.clock()
print output_prob
print 'runtime is: %f ms.'%((end-start)*1000)
```

另外值得注意的一点是，由于我们的caffe是基于C++训练的，所以均值文件类型为binaryproto，而pycaffe需要的均值文件类型为numpy，这个转换可以通过一个简单的脚本实现：

```
import caffe
import numpy as np

MEAN_PROTO_PATH = './myimagenet_mean.binaryproto'
MEAN_NPY_PATH = './mean.npy'

blob = caffe.proto.caffe_pb2.BlobProto()
data = open(MEAN_PROTO_PATH, 'rb' ).read()
blob.ParseFromString(data)

array = np.array(caffe.io.blobproto_to_array(blob))
mean_npy = array[0]
np.save(MEAN_NPY_PATH ,mean_npy)
```

最后给出CPU和GPU的实际运行截图：

**Caffe - CPU:**

```
nvidia@tegra-ubuntu: ~/caffemodel
I0627 13:06:06.054478 9652 net.cpp:200] pool2 does not need backward computation.
I0627 13:06:06.054518 9652 net.cpp:200] relu2 does not need backward computation.
I0627 13:06:06.054534 9652 net.cpp:200] conv2 does not need backward computation.
I0627 13:06:06.054553 9652 net.cpp:200] norm1 does not need backward computation.
I0627 13:06:06.054570 9652 net.cpp:200] pool1 does not need backward computation.
I0627 13:06:06.054586 9652 net.cpp:200] relu1 does not need backward computation.
I0627 13:06:06.054603 9652 net.cpp:200] conv1 does not need backward computation.
I0627 13:06:06.054620 9652 net.cpp:200] data does not need backward computation.
I0627 13:06:06.054636 9652 net.cpp:242] This network produces output prob
I0627 13:06:06.054687 9652 net.cpp:255] Network initialization done.
I0627 13:06:06.381431 9652 net.cpp:744] Ignoring source layer loss
img path: '/usr/src/tensorrt/samples/giexec/t2.jpeg'
[ 97.25115522 108.84620235 116.72652473]
[4.9393061e-06 9.9999511e-01 1.3395301e-08]
runtime is: 759.026000 ms.
nvidia@tegra-ubuntu:~/caffemodel$
```

Caffe - GPU:

```
nvidia@tegra-ubuntu: ~/caffemodel
I0627 13:02:47.788849 9480 net.cpp:200] pool2 does not need backward computation.
I0627 13:02:47.788866 9480 net.cpp:200] relu2 does not need backward computation.
I0627 13:02:47.788914 9480 net.cpp:200] conv2 does not need backward computation.
I0627 13:02:47.788962 9480 net.cpp:200] norm1 does not need backward computation.
I0627 13:02:47.788982 9480 net.cpp:200] pool1 does not need backward computation.
I0627 13:02:47.788998 9480 net.cpp:200] relu1 does not need backward computation.
I0627 13:02:47.789014 9480 net.cpp:200] conv1 does not need backward computation.
I0627 13:02:47.789029 9480 net.cpp:200] data does not need backward computation.
I0627 13:02:47.789044 9480 net.cpp:242] This network produces output prob
I0627 13:02:47.789099 9480 net.cpp:255] Network initialization done.
I0627 13:02:48.117187 9480 net.cpp:744] Ignoring source layer loss
img path: '/home/nvidia/Downloads/train/label_3/p2_img116_0198_2_rotate.jpeg'
[ 97.25115522 108.84620235 116.72652473]
[9.9962676e-01 6.4648600e-07 3.7264705e-04]
runtime is: 57.834000 ms.
nvidia@tegra-ubuntu:~/caffemodel$
```

## 4.tensorRT+caffe

终于到了最激动人心的环节了：利用tensorRT加速caffe的运行，具体加速效果卖个关子，放到最后展示。

不过首先我要声明一件事，那就是tensorRT + caffe其实和caffe一点关系都没有。tensorRT自带了**CaffeParser**，可以读取caffe模型的prototxt文件并针对性的进行优化，无需搭建caffe环境。

前面教你装caffe其实就只是为了看看未加速之前的效果，嘻嘻。

声明：以下tensorRT采用C++实现，python版本tensorRT也存在，这里不作演示。

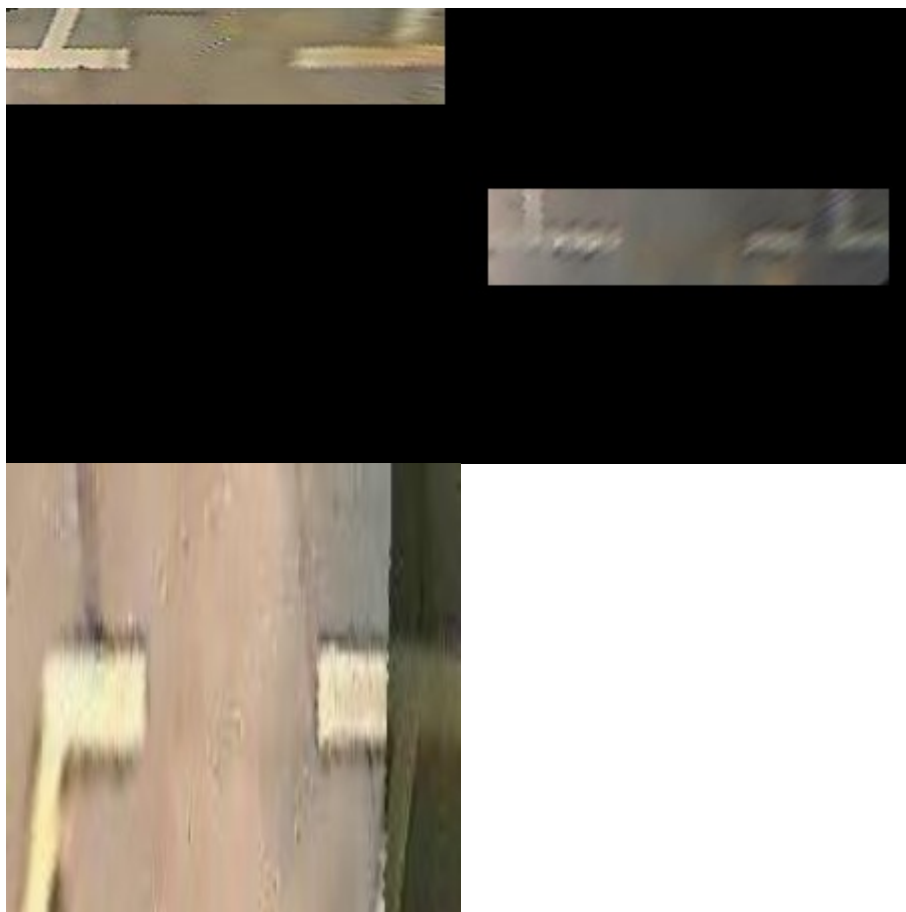
下面进入正题，如何使用tensorRT：

首先说一下我在实现过程中遇到的一些坑：

- 如何读取图片：caffe网络层接收的输入为一个大小为3x227x227的**一维数组**，如何将图片转换为一维数组是第一个难点。
- 图片格式如何处理：类似于pycaffe，即使你想办法读取到了图片，图片的存储格式可能是CxWxH，WxHxC，CxHxW等很多不同的格式，而且C内部还可能是RGB或者BGR，如何将读取的图片转换成正确的格式输入给网络也是一个很大的难点。
- 图片resize方法问题：由于caffe网络的输入层定义如下：

```
layer {
  name: "data"
  type: "Input"
  top: "data"
  input_param { shape: { dim: 10 dim: 3 dim: 227 dim: 227 } }
}
```

可以看到，Input也就是输入层的数据格式为**3x227x227**，最前面的10是batch size，这里没有影响。然而输入图片大部分为长方形的库位点对，大小大概在**210x48**左右，所以需要将输入图片resize之后再放进输入层。这就涉及到如何resize的问题：是原图在正中间周围加padding，还是原图在左上角右下角加padding，还是单纯的直接图片拉伸。





如图所示为三种不同的resize方法。

- 均值文件问题：这个放到后面讲。

下面逐个解释如何解决这些问题：

- 读取图片：输入图片为jpeg格式，在没有openCV或任何库的加持下（tensorRT并不自带读取图片的函数），恐怕难以实现。因此，我采用了**stb\_image**这个小巧的库，可以直接读取jpeg格式的图片。（jpeg和jpg只是同一种东西的不同叫法，其实是一模一样的）。**但是!!!**经过我的实际验证，stb\_image读取的图片格式是**CxWxH**，雪上加霜的是，C还是**RGB**！而caffe的输入要求为**WxHxC**，C为**BGR**。这就很尴尬了，读完了还得手动转换格式。

- 转换图片格式：C(RGB)xWxH的格式看起来应该是这样子的：RGBRGBRGBRGBRGBRGBRGBRGB...

而WxHxC(BGR)的格式看起来是这样子：BBBBBBBBBB...GGGGGGGGGG...RRRRRRRRRR

你是不是开始晕了？呵呵，其实在我没摸透格式之前还试过CxHxW.....

不过既然我已经明白白的告诉你格式了，转换应该不是一件困难的事情了，困难的是摸透格式之前的各种尝试。为了知道stb\_image的读取格式，我把图片用matlab读取，然后尝试了各种组合方法才试出来，caffe的输入格式也是费了很大劲才匹配到。

具体实现方法请参考源码giexec.cpp。

- 图片resize：没什么好说的，各种resize方法试一遍，最后发现其实caffe是直接把原图给拉伸了。另外根据我的研究表明，BGR通道不小心弄成RGB的话，对结果不会有非常大的影响，概率从1变成0.9这样子。当然我们知道了格式是BGR，肯定按BGR来了。
- 均值文件问题：这个问题也是由于我对caffe不太熟悉吧，最后我才得知，RGB图片的均值是每个通道分别求均值，所以均值文件应该是**三个数**，代表**RGB三个通道**的平均值，减去均值的时候要将每个通道的所有的点减去其对应的均值。（这个均值我直接从**pycaffe**里得到了，在C++的**binaryproto**中没有直接给出这三个数，需要进一步计算）。

代码实现参考了自带的**sampleMNIST**的例子，由于我们的caffe模型读取的不是二值图片，所以在以上问题上要注意坑点，其他部分与例子结构是大体相似的，因为tensorRT的执行流程是一样的。下面给出代码：

```
#include <assert.h>
#include <fstream>
#include <sstream>
#include <iostream>
#include <cmath>
#include <algorithm>
#include <sys/stat.h>
#include <time.h>
#include <cuda_runtime_api.h>

#include "NvInfer.h"
#include "NvCaffeParser.h"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"

using namespace nvinfer1;
using namespace nvcaffeparser1;

#define CHECK(status) \
```

[illegible]

```

    // specify which tensors are outputs
    for (auto& s : outputs)
        network->markOutput(*blobNameToTensor->find(s.c_str()));

    // Build the engine
    builder->setMaxBatchSize(maxBatchSize);
    builder->setMaxWorkspaceSize(1 << 20);

    ICudaEngine* engine = builder->buildCudaEngine(*network);
    assert(engine);

    // we don't need the network any more, and we can destroy the parser
    network->destroy();
    parser->destroy();

    // serialize the engine, then close everything down
    gieModelStream = engine->serialize();
    engine->destroy();
    builder->destroy();
    shutdownProtobufLibrary();
}

void doInference(IExecutionContext& context, float* input, float* output, int batchSize)
{
    const ICudaEngine& engine = context.getEngine();
    // input and output buffer pointers that we pass to the engine - the engine requires exactly
    IEngine::getNbBindings(),
    // of these, but in this case we know that there is exactly one input and one output.
    assert(engine.getNbBindings() == 2);
    void* buffers[2];

    // In order to bind the buffers, we need to know the names of the input and output tensors.
    // note that indices are guaranteed to be less than IEngine::getNbBindings()
    int inputIndex = engine.getBindingIndex(INPUT_BLOB_NAME),
        outputIndex = engine.getBindingIndex(OUTPUT_BLOB_NAME);

    // create GPU buffers and a stream
    CHECK(cudaMalloc(&buffers[inputIndex], batchSize * INPUT_D * INPUT_H * INPUT_W *
sizeof(float)));
    CHECK(cudaMalloc(&buffers[outputIndex], batchSize * OUTPUT_SIZE * sizeof(float)));

    cudaStream_t stream;
    CHECK(cudaStreamCreate(&stream));
    cudaEvent_t start, end; //calculate run time
    CHECK(cudaEventCreate(&start));
    CHECK(cudaEventCreate(&end));

    // DMA the input to the GPU, execute the batch asynchronously, and DMA it back:
    float ms;
    CHECK(cudaMemcpyAsync(buffers[inputIndex], input, batchSize * INPUT_D * INPUT_H * INPUT_W *
sizeof(float), cudaMemcpyHostToDevice, stream));
    cudaEventRecord(start, stream);

```

```

    context.enqueue(batchSize, buffers, stream, nullptr);
    CHECK(cudaMemcpyAsync(output, buffers[outputIndex], batchSize * OUTPUT_SIZE * sizeof(float),
        cudaMemcpyDeviceToHost, stream));
    cudaEventRecord(end, stream);
    cudaEventSynchronize(end);
    cudaEventElapsedTime(&ms, start, end);
    cudaStreamSynchronize(stream);
    cudaEventDestroy(start);
    cudaEventDestroy(end);
    std::cout<<"execution time:"<<ms<<"ms."<<std::endl;
    // release the stream and the buffers
    cudaStreamDestroy(stream);
    CHECK(cudaFree(buffers[inputIndex]));
    CHECK(cudaFree(buffers[outputIndex]));
}

```

```

int main(int argc, char** argv)
{
    // create a GIE model from the caffe model and serialize it to a stream
    IHostMemory *gieModelStream{nullptr};
    std::cout<<"Converting from caffe model..."<<std::endl;
    caffeToGIEModel("/home/nvidia/caffemodel/deploy.prototxt",
        "/home/nvidia/caffemodel/mycaffenet_train_iter_450000.caffemodel", std::vector < std::string > {
        OUTPUT_BLOB_NAME }, 1, gieModelStream);
    std::cout<<"Conversion successful!"<<std::endl;
    unsigned char* fileData;
    std::cout<<"Reading jpg image..."<<std::endl;
    std::string fname;
    std::cout<<"Input file name:"<<std::endl;
    std::cin>>fname;
    readJPG(fname, fileData);
    std::cout<<"Reading successful!"<<std::endl;

    std::cout<<"Converting data..."<<std::endl;
    unsigned char* newData = (unsigned char*)malloc(INPUT_D*INPUT_H*INPUT_W*sizeof(unsigned
        char));
    int start = 2;
    for(int i=0;i<227*227;i++)
    {
        newData[i] = fileData[start]-97.25115522;    //magic, don't modify!
        start += 3;
    }
    start = 1;
    for(int i=227*227;i<227*227*2;i++)
    {
        newData[i] = fileData[start]-108.84620235;    //magic, don't modify!
        start += 3;
    }
    start = 0;
    for(int i=227*227*2;i<227*227*3;i++)
    {
        newData[i] = fileData[start]-116.72652473;    //magic, don't modify!
    }
}

```

```

        start += 3;
    }
    float data[INPUT_D*INPUT_H*INPUT_W];
    for (int i = 0; i < INPUT_D*INPUT_H*INPUT_W; i++)
        data[i] = float(newData[i]);
    std::cout<<"Converted."<<std::endl;

    std::cout<<"Deserializing the engine..."<<std::endl;
    // deserialize the engine
    IRuntime* runtime = createInferRuntime(gLogger);
    ICudaEngine* engine = runtime->deserializeCudaEngine(gieModelStream->data(), gieModelStream->size(), nullptr);
    if (gieModelStream) gieModelStream->destroy();

    IExecutionContext *context = engine->createExecutionContext();
    std::cout<<"Deserialized."<<std::endl;
    // run inference
    float prob[OUTPUT_SIZE];
    std::cout<<"Running inference..."<<std::endl;
    doInference(*context, data, prob, 1);
    std::cout<<"Inference successful!"<<std::endl;
    // destroy the engine
    context->destroy();
    engine->destroy();
    runtime->destroy();
    // print a histogram of the output distribution
    for (unsigned int i = 0; i < 3; i++)
    {
        std::cout<<prob[i]<<std::endl;
    }
    return 0;
}

```

那么，加速后的caffe模型的inference能快多少倍呢？

答案是**整整五倍**！运行时间从50ms左右降低到了10ms左右，性能提升还是非常可观的。

与CPU版本的pyCaffe相比，从800ms左右降低到了10ms，提升了**八十倍**！。

由此可以看出，tensorRT的加速能力十分出众，因此接下来我们将它运用到了YOLO上，只是很可惜没有完全成功。

```
nvidia@tegra-ubuntu: /usr/src/tensorrt/samples/giexec
Compiling: giexec.cpp
Linking: ../../bin/giexec_debug
Compiling: giexec.cpp
Linking: ../../bin/giexec
nvidia@tegra-ubuntu:/usr/src/tensorrt/samples/giexec$ sudo gedit giexec.cpp
** (gedit:9599): WARNING **: Set document metadata failed: Setting attribute met
adata::gedit-position not supported
nvidia@tegra-ubuntu:/usr/src/tensorrt/samples/giexec$ ../../bin/giexec
Converting from caffe model...
Conversion successful!
Reading jpg image...
Reading successful!
Converting data...
Converted.
Deserializing the engine...
Deserialized.
Running inference...
execution time:9.20976ms.
Inference successful!
0.999995
1.82439e-08
4.94933e-06
nvidia@tegra-ubuntu:/usr/src/tensorrt/samples/giexec$
```

## 5.tensorRT+YOLO

由于YOLO的网络结构比较特殊（tensorRT比较辣鸡），tensorRT并不能直接加速YOLO的目标检测。所以，为了加速YOLO的运行，需要先将其转换为caffe网络结构，因为tensorRT对caffe，tensorflow等主流深度学习框架支持比较完善。

转换YOLO的过程我们用到了一个转换工具，可以支持pytorch-YOLO-caffe互相转换，由于YOLO->caffe的过程也依赖与pytorch，所以索性一起把pytorch的环境也配好了。

## 6.pytorch配置

然而，TX2的CPU架构为AArch64，并不是一般台式机的架构，这种架构常见于安卓智能机。因此，常用的pytorch在ubuntu上的配置教程是不适用的。在这里笔者参考了NVIDIA在GitHub上给出的配置教程，给出可行的解决方案：

首先，后面会用到cmake，然而NVIDIA的教程并没有讲明，所以先装上：

```
sudo apt-get install cmake
```

接下来就是按照教程进行了。经历了caffe的配置过程，你应该已经有了pip和git，这里就不再重复，直接进入正题：

```
# clone pyTorch repo
git clone http://github.com/pytorch/pytorch

cd pytorch
```

```
git submodule update --init

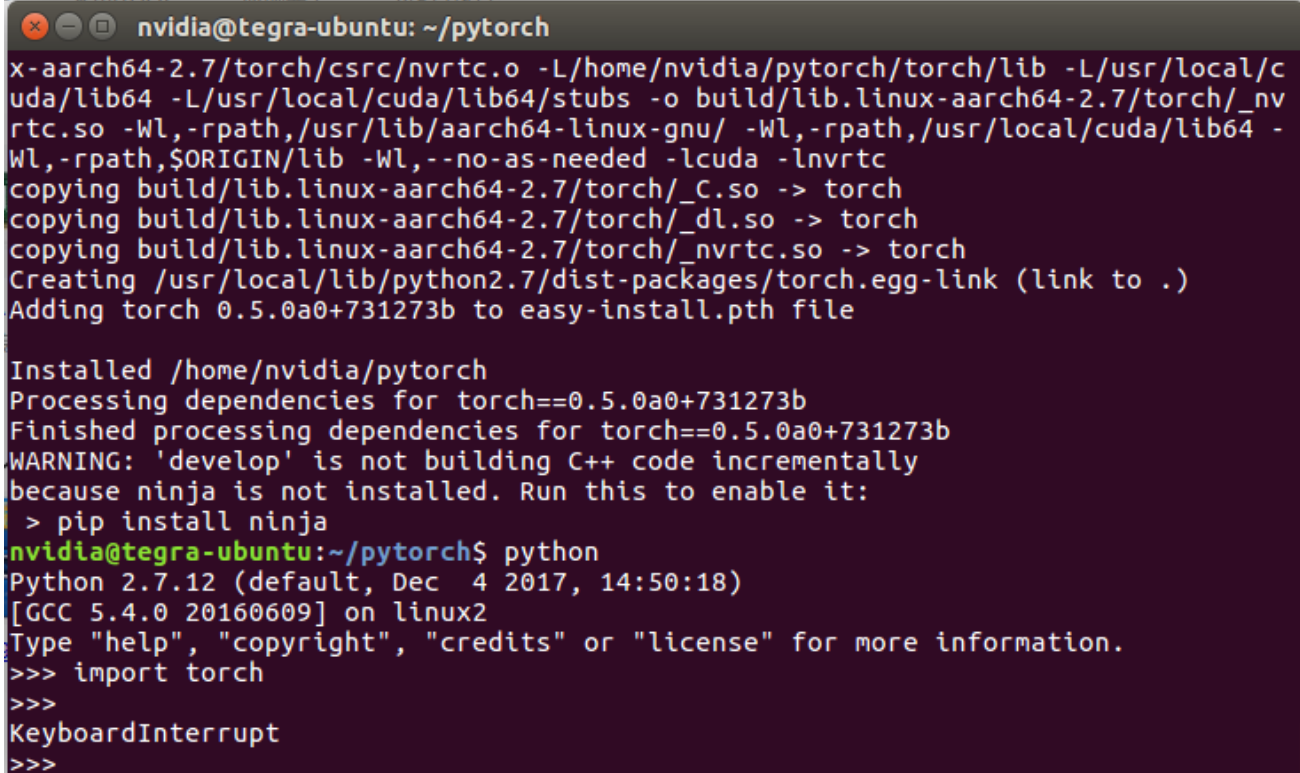
# install prereqs
sudo pip install -U setuptools
sudo pip install -r requirements.txt

# Develop Mode:
python setup.py build_deps
sudo python setup.py develop
```

**注意，中间两条pip install指令可以按caffe教程中设置为清华源，速度奇快。**

经历了漫长的过程以后，torch应该已经安装好了，这时可以用python试验一下：

```
python
import torch
```



```
nvidia@tegra-ubuntu: ~/pytorch
x-aarch64-2.7/torch/csrc/nvrtc.o -L/home/nvidia/pytorch/torch/lib -L/usr/local/cuda/lib64 -L/usr/local/cuda/lib64/stubs -o build/lib.linux-aarch64-2.7/torch/_nvrtc.so -Wl,-rpath,/usr/lib/aarch64-linux-gnu/ -Wl,-rpath,/usr/local/cuda/lib64 -Wl,-rpath,$ORIGIN/lib -Wl,--no-as-needed -lcuda -lnvrtc
copying build/lib.linux-aarch64-2.7/torch/_C.so -> torch
copying build/lib.linux-aarch64-2.7/torch/_dl.so -> torch
copying build/lib.linux-aarch64-2.7/torch/_nvrtc.so -> torch
Creating /usr/local/lib/python2.7/dist-packages/torch.egg-link (link to .)
Adding torch 0.5.0a0+731273b to easy-install.pth file

Installed /home/nvidia/pytorch
Processing dependencies for torch==0.5.0a0+731273b
Finished processing dependencies for torch==0.5.0a0+731273b
WARNING: 'develop' is not building C++ code incrementally
because ninja is not installed. Run this to enable it:
> pip install ninja
nvidia@tegra-ubuntu:~/pytorch$ python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>>
KeyboardInterrupt
>>>
```

如果什么都没发生，说明安装成功，恭喜。

## 7.Darknet to caffe

如果你已经搭好了pytorch和caffe的环境，下一步就是将已有的YOLOv2模型转换成caffe模型了。

为了实现这一过程，需要用到一个转换工具：<https://github.com/ysh329/darknet2caffe>

在所给链接下载所需文件解压后，进入目录。



```
cd darknet2caffe-master
```

将自己的YOLO模型的.cfg和.weights文件也拷贝到当前目录。（这个不用教了吧）

接下来，先别急着跑py文件，还记得你的caffe怎么用的吗？

对，还得先设置一下caffe的环境变量，不然无法import。当然如果你有更好的办法永久将caffe加入python的环境变量，这一步可以跳过。

```
export PYTHONPATH=/home/nvidia/caffe-master/python:$PYTHONPATH
```

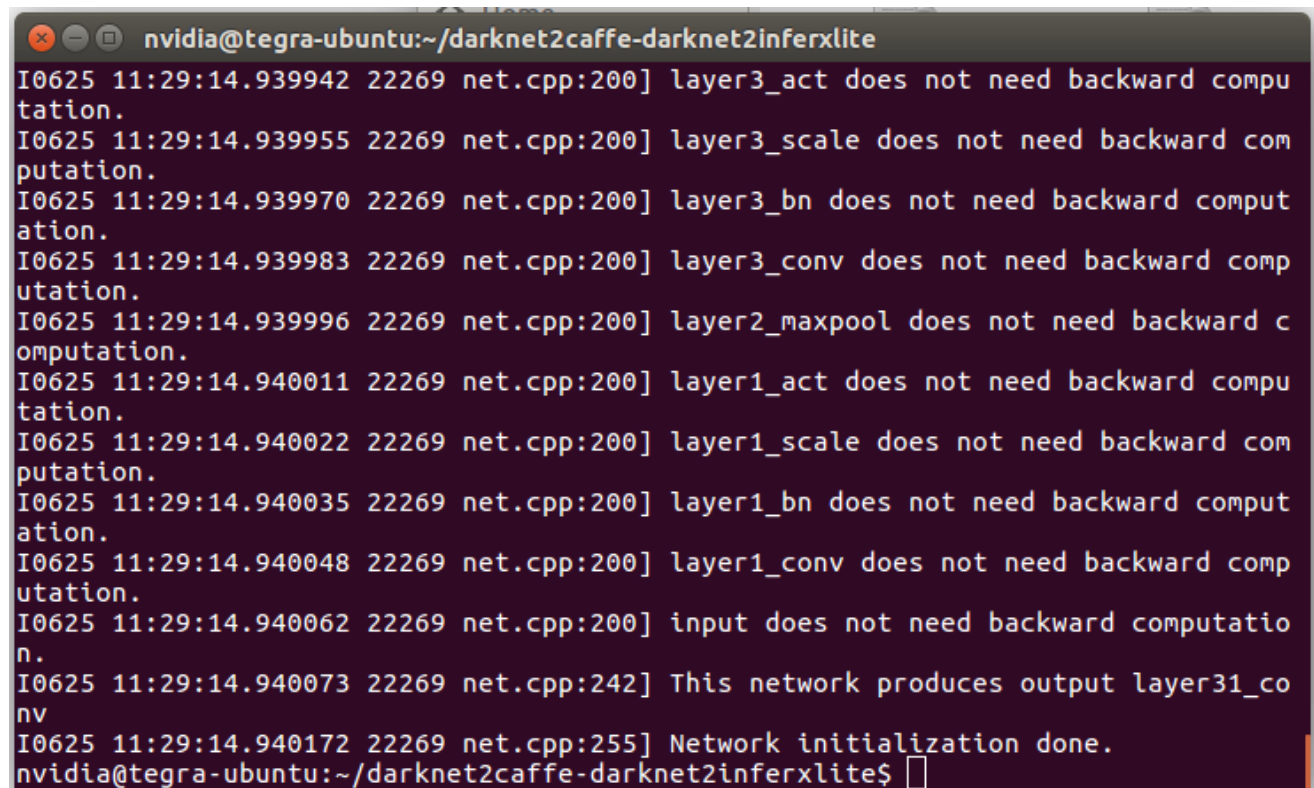
命令行变白以后，可以执行python脚本了：

```
python darknet2caffe.py DARKNET_CFG DARKNET_WEIGHTS
```

两个参数分别为YOLOv2的cfg文件和weights文件的文件名。

执行命令以后，最后会出现如下输出：

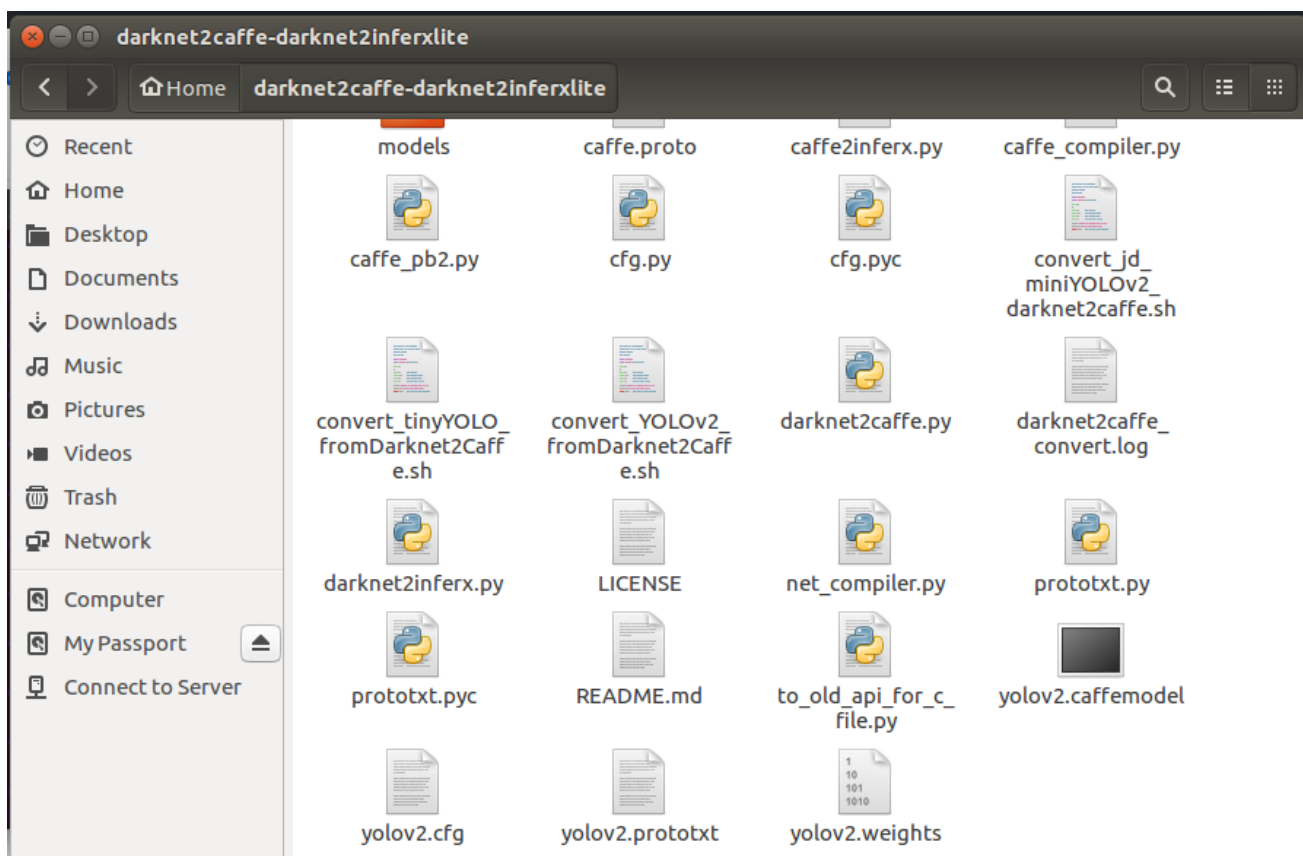
```
Network initialization done.
```



```
nvidia@tegra-ubuntu:~/darknet2caffe-darknet2inferxlite
I0625 11:29:14.939942 22269 net.cpp:200] layer3_act does not need backward computation.
I0625 11:29:14.939955 22269 net.cpp:200] layer3_scale does not need backward computation.
I0625 11:29:14.939970 22269 net.cpp:200] layer3_bn does not need backward computation.
I0625 11:29:14.939983 22269 net.cpp:200] layer3_conv does not need backward computation.
I0625 11:29:14.939996 22269 net.cpp:200] layer2_maxpool does not need backward computation.
I0625 11:29:14.940011 22269 net.cpp:200] layer1_act does not need backward computation.
I0625 11:29:14.940022 22269 net.cpp:200] layer1_scale does not need backward computation.
I0625 11:29:14.940035 22269 net.cpp:200] layer1_bn does not need backward computation.
I0625 11:29:14.940048 22269 net.cpp:200] layer1_conv does not need backward computation.
I0625 11:29:14.940062 22269 net.cpp:200] input does not need backward computation.
I0625 11:29:14.940073 22269 net.cpp:242] This network produces output layer31_conv
I0625 11:29:14.940172 22269 net.cpp:255] Network initialization done.
nvidia@tegra-ubuntu:~/darknet2caffe-darknet2inferxlite$
```

这说明转换成功，在当前目录下会出现.prototxt和.caffemodel两个文件。如图，笔者的是yolov2.prototxt和yolov2.caffemodel。





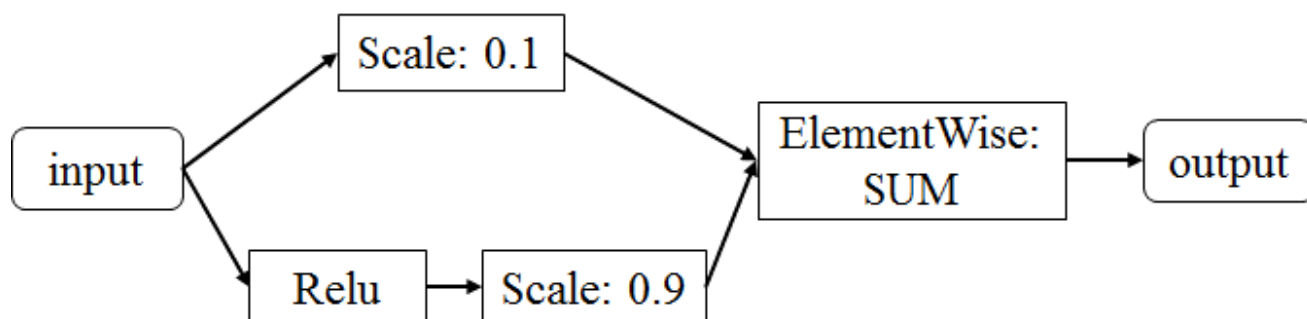
## 8. Try to run YOLO with tensorRT

首先我得承认，用tensorRT优化YOLOv2的计划并没有完全成功，我们卡在了导入模型这一步上。

原因有二：一是tensorRT的**CaffeParser**不支持**LeakyReLU**层，二是YOLOv2新增了最后的**region**层，这也是caffe中所没有的。

经过我们在网上的深度调研，得出了以下结论：

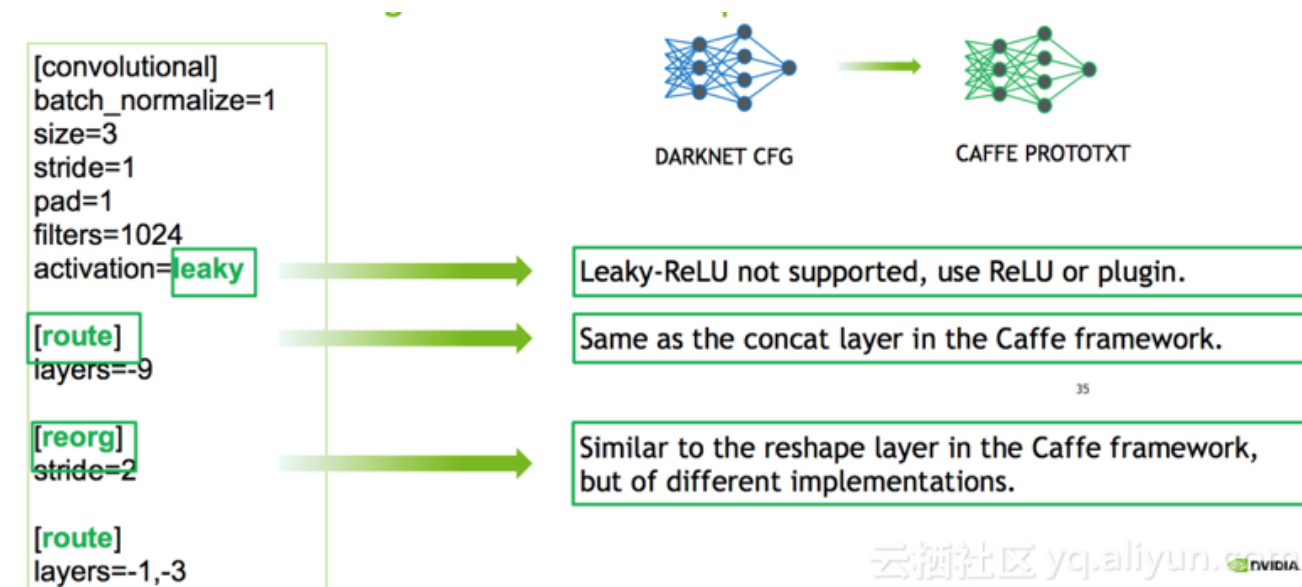
- 针对**LeakyReLU**不支持的问题，在NVIDIA官方论坛上的大佬给出了解决方案：用**standard-ReLU + scale + eltwise** 来近似**leakyReLU**。



这种方法貌似可以通过直接修改prototxt中的layer结构实现，然而我们时间有限，并没有机会去尝试。

- 针对**region**层不支持的问题，我们采用了折衷方案，打算采用**tensorRT + YOLOv1**的方案实现。因为YOLOv1是没有region层的。然而v1也存在leakyReLU的问题，所以也没有实际测试效果。
- 如果非要用**v2**，可以把网络拆开，从倒数第二层输出结果，自己用代码实现最后一层region层的功能，不过这个方法的难度又更上一层楼了，所以我们也没有机会尝试。

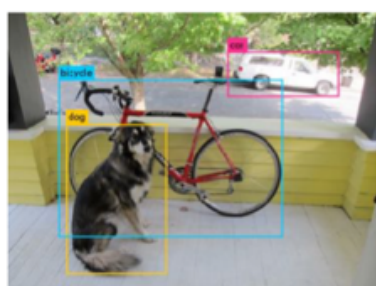
然而，在一场NVIDIA高级工程师的直播中，提到了利用tensorRT加速YOLOv2的例子，并且声称可以加速到原来的5倍左右。然而这场直播的录播中，YOLOv2相关部分被快进了，所以具体操作方法我们也不得而知了。。。



上图是直播中的简要介绍，可以看到思路和我上面讲的差不多，但是具体实现起来我们还是没有头绪和精力。

## YOLOv2 with INT8

### Performance with INT8 acceleration



Darknet cuDNN  
Perf: 11.3ms



TensorRT FP32  
Perf: 4.80ms



TensorRT INT8  
Perf: 2.34ms

云栖社区 yq.aliyun.com

这张是他们做的结果，可以看到INT8推断比最开始的Darknet快了5倍，但是具体他是怎么做的直播里面被快进了啊啊啊啊啊。。。好吧，其实我有联系过直播的主讲人，NVIDIA的高级工程师，然而人家日理万机，并没有功夫理我这个菜鸟。

下面是直播地址，虽然YOLO部分快进了，但是前面介绍tensorRT的部分还是值得一看的。

<https://yq.aliyun.com/video/play/1381?spm=a2c4e.11153940.blogcont580307.10.797a331bccbnye>

进去链接在右边选中间的章节，1小时的那个视频。

由于v1只有LeakyReLU的问题，我们尝试了直接把LeakyReLU改成ReLU然后用tensorRT模拟加速（随机输入数据），得出的结果是每张图片要花大概0.04s去推断，加速之前是0.4s，加速效果还是很理想的。按这样计算，加速后的FPS可以达到25左右，接近实时的效果。

YOLOv1

```

nvidia@tegra-ubuntu: ~/darknet-master
19 max      2 x 2 / 2    28 x 28 x1024 -> 14 x 14 x1024
20 conv     512 1 x 1 / 1    14 x 14 x1024 -> 14 x 14 x 512  0.206 BFL
OPs
21 conv    1024 3 x 3 / 1    14 x 14 x 512 -> 14 x 14 x1024  1.850 BFL
OPs
22 conv     512 1 x 1 / 1    14 x 14 x1024 -> 14 x 14 x 512  0.206 BFL
OPs
23 conv    1024 3 x 3 / 1    14 x 14 x 512 -> 14 x 14 x1024  1.850 BFL
OPs
24 conv    1024 3 x 3 / 1    14 x 14 x1024 -> 14 x 14 x1024  3.699 BFL
OPs
25 conv    1024 3 x 3 / 2    14 x 14 x1024 -> 7 x 7 x1024  0.925 BFL
OPs
26 conv    1024 3 x 3 / 1    7 x 7 x1024 -> 7 x 7 x1024  0.925 BFL
OPs
27 conv    1024 3 x 3 / 1    7 x 7 x1024 -> 7 x 7 x1024  0.925 BFL
OPs
28 Local Layer: 7 x 7 x 1024 image, 256 filters -> 7 x 7 x 256 image
29 dropout      p = 0.50      12544 -> 12544
30 connected      12544 -> 784
31 Detection Layer
forced: Using default '0'
Loading weights from yolov1_3000.weights...Done!
imgs/001.jpg: Predicted in 0.464953 seconds.

```

## YOLOv1 - tensorRT

```

nvidia@tegra-ubuntu: /usr/src/tensorrt/samples/giexec
'/home/nvidia/darknet-master/yolov1.prototxt' --half2=true
deploy: /home/nvidia/darknet-master/yolov1.prototxt
half2
At least one network output must be defined
nvidia@tegra-ubuntu: /usr/src/tensorrt/samples/giexec$ ../../bin/giexec --deploy=
'/home/nvidia/darknet-master/yolov1.prototxt' --half2=true --output=layer31_fc
deploy: /home/nvidia/darknet-master/yolov1.prototxt
half2
output: layer31_fc
Input "data": 3x448x448
Output "layer31_fc": 784x1x1
name=data, bindingIndex=0, buffers.size()=2
name=layer31_fc, bindingIndex=1, buffers.size()=2
Average over 10 runs is 37.6406 ms.
Average over 10 runs is 37.6132 ms.
Average over 10 runs is 37.5364 ms.
Average over 10 runs is 37.6929 ms.
Average over 10 runs is 37.7098 ms.
Average over 10 runs is 37.4753 ms.
Average over 10 runs is 37.6028 ms.
Average over 10 runs is 37.6275 ms.
Average over 10 runs is 37.5916 ms.
Average over 10 runs is 37.626 ms.
nvidia@tegra-ubuntu: /usr/src/tensorrt/samples/giexec$ 

```

然而这只是模拟运行，还是把leakyReLU直接偷懒改掉了，实际运行结果肯定会有偏差，所以这里只是写上作为参考。另外，v1的权重文件太大了，700多M，过于笨重。

因此，最后我们决定暂时搁置对YOLO的加速，转而直接使用了**tiny YOLOv2**，这是YOLOv2的轻量化版本，经过我们的训练权重文件只有几十M，比v1要小十几倍。而且经过测试，正常情况下的车位点识别情况也比较理想。

最关键的是，未经加速的tiny YOLOv2的理论FPS就可以达到30左右！所以对于实时性而言，tiny YOLOv2基本满足需求。

## YOLOv2 - CPU

```
nvidia@tegra-ubuntu: ~/darknet-master
OPs
 23 conv  1024  3 x 3 / 1   13 x  13 x1024  ->   13 x  13 x1024  3.190 BFL
OPs
 24 conv  1024  3 x 3 / 1   13 x  13 x1024  ->   13 x  13 x1024  3.190 BFL
OPs
 25 route  16
 26 conv   64  1 x 1 / 1   26 x  26 x 512  ->   26 x  26 x  64  0.044 BFL
OPs
 27 reorg
 28 route  27 24
 29 conv  1024  3 x 3 / 1   13 x  13 x1280  ->   13 x  13 x1024  3.987 BFL
OPs
 30 conv   30  1 x 1 / 1   13 x  13 x1024  ->   13 x  13 x  30  0.010 BFL
OPs
 31 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2.weights...Done!
imgs/001.jpg: Predicted in 9.244729 seconds.
person: 79%
person: 75%
person: 69%
person: 62%
person: 54%
nvidia@tegra-ubuntu:~/darknet-master$
```

## YOLOv2

```
nvidia@tegra-ubuntu: ~/darknet-master
OPs
 23 conv  1024  3 x 3 / 1   13 x  13 x1024  ->   13 x  13 x1024  3.190 BFL
OPs
 24 conv  1024  3 x 3 / 1   13 x  13 x1024  ->   13 x  13 x1024  3.190 BFL
OPs
 25 route  16
 26 conv   64  1 x 1 / 1   26 x  26 x 512  ->   26 x  26 x  64  0.044 BFL
OPs
 27 reorg
 28 route  27 24
 29 conv  1024  3 x 3 / 1   13 x  13 x1280  ->   13 x  13 x1024  3.987 BFL
OPs
 30 conv   30  1 x 1 / 1   13 x  13 x1024  ->   13 x  13 x  30  0.010 BFL
OPs
 31 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2.weights...Done!
imgs/001.jpg: Predicted in 0.117505 seconds.
person: 79%
person: 75%
person: 69%
person: 62%
person: 54%
nvidia@tegra-ubuntu:~/darknet-master$
```

## YOLOv2 - tiny

```

nvidia@tegra-ubuntu: ~/darknet-master
 6 conv      128  3 x 3 / 1   52 x  52 x  64  ->  52 x  52 x 128  0.399 BFL
OPs
 7 max
 8 conv      256  3 x 3 / 1   26 x  26 x 128  ->  26 x  26 x 256  0.399 BFL
OPs
 9 max
10 conv      512  3 x 3 / 1   13 x  13 x 256  ->  13 x  13 x 512  0.399 BFL
OPs
11 max
12 conv     1024  3 x 3 / 1   13 x  13 x 512  ->  13 x  13 x1024  1.595 BFL
OPs
13 conv     1024  3 x 3 / 1   13 x  13 x1024  ->  13 x  13 x1024  3.190 BFL
OPs
14 conv       30  1 x 1 / 1   13 x  13 x1024  ->  13 x  13 x  30  0.010 BFL
OPs
15 detection
mask_scale: Using default '1.000000'
Loading weights from yolov2-t.weights...Done!
imgs/001.jpg: Predicted in 0.035949 seconds.
person: 83%
person: 82%
person: 82%
person: 69%
nvidia@tegra-ubuntu:~/darknet-master$

```

最后奉上各模型运行速度总结：

库位点检测：

NETWORK	SPEED
YOLOv2 - CPU	9000ms
YOLOv1 - GPU	400ms
YOLOv2 - GPU	100ms
YOLOv1 - tensorRT	37ms
YOLOv2 - tiny	35ms

库位分类：

NETWORK	SPEED
pyCaffe - CPU	759ms
pyCaffe - GPU	57ms
Caffe - tensorRT	9ms

其实最开始我编译darknet的时候忘了把GPU和cuDNN改成1，所以跑了9秒钟，当时简直怀疑人生，没想到最后能加速到35ms，足足翻了257倍的速度，现在回想起来也真的是一步步在进步，收获颇丰。

同济大学软件学院 Dan