Caffe

学习笔记

**甄海华**

**2019年**

内容目录

[1. Windows平台下Caffe的安装 3](#_Toc22493091)

[2. Windows下编译及调试Debug版Caffe 7](#_Toc22493092)

[3. Caffe源码函数调用关系图 13](#_Toc22493093)

[4. Caffe中Solver、Net、Layer、Blob的构建流程 14](#_Toc22493094)

[5. Windows训练与测试Caffe mnist 16](#_Toc22493095)

[6. Caffe solver文件参数详细解析 18](#_Toc22493096)

[7. Windows使用Cifar10训练及测试Caffe版DenseNet 20](#_Toc22493097)

[8. Caffe py-Faster-RCNN 源码解析 29](#_Toc22493098)

[9. Caffe 添加自定义层：C++层 36](#_Toc22493099)

[10. Caffe添加自定义层：Python层 42](#_Toc22493100)

1. Windows平台下Caffe的安装

系统环境：Windows 10系统+CUDA

按照以下教程安装：

手把手从0开始安装Windows版Caffe与py-faster-RCNN（https://blog.csdn.net/AManFromEarth/article/details/80212554）

安装期间出现以上教程没有出现的问题，如下：

（1） VS Build libcaffe 报错

error MSB4062: 未能从程序集 E:\NugetPackages\OpenCV.2.4.10\......的解决办法

解决方法：

VS——项目——管理NuGet程序包——更新——OpenCV，选择更新（只用更新OpenCV，其他不用更新），更新后修改packages.config（VS会自动修改），并重新加载（VS会自动加载），加载后再次build libcaffe。

（2） VS Build libCaffe报错：

1>C:\zhh\NugetPackages\OpenCV.2.4.10\build\native\OpenCV.targets(772,5): error : NuGet Error:操作已超时

解决方法：

公司网络使用代理上外网，无法使用代理服务器或公司网络链接，改为用wifi连接手机热点，再build

（3）VS Build libCaffe报错：

error MSB3721: 命令“"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\bin\nvcc.exe" -gencode=arch=compute\_61,code=\"sm\_61,compute\_61\" --use-local-env --cl-version 2013 -ccbin "C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin\x86\_amd64"  -IC:\zhh\NugetPackages\OpenCV.2.4.11\build\native\../../build/native/include/ -I"C:\zhh\NugetPackages\lmdb-v120-clean.0.9.14.0\build\native\..\..\lib\native\include" -I"C:\zhh\NugetPackages\LevelDB-vc120.1.2.0.0\build\native\../..//build/native/include/" -I"C:\zhh\NugetPackages\protobuf-v120.2.6.1\build\native\../..//build/native/include/" -IC:\zhh\NugetPackages\glog.0.3.3.0\build\native\../..//build/native/include/ -IC:\zhh\NugetPackages\gflags.2.1.2.1\build\native\../..///build/native/include/ -IC:\zhh\NugetPackages\boost.1.59.0.0\build\native\..\..\lib\native\include\ -I"C:\zhh\NugetPackages\hdf5-v120-complete.1.8.15.2\build\native\..\..\lib\native\include" -IC:\zhh\NugetPackages\OpenBLAS.0.2.14.1\build\native\..\..\lib\native\include -I"C:\zhh\caffe-master\windows\libcaffe\\..\..\src\\" -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include" -I"C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v8.0\include"     --keep-dir C:\zhh\caffe-master\windows\..\Build\Int\libcaffe\x64\Release -maxrregcount=0  --machine 64 --compile -cudart static -Xcudafe "--diag\_suppress=exception\_spec\_override\_incompat --diag\_suppress=useless\_using\_declaration --diag\_suppress=field\_without\_dll\_interface" -D\_SCL\_SECURE\_NO\_WARNINGS -DGFLAGS\_DLL\_DECL=    -DHAS\_OPENCV -DHAS\_LMDB -DHAS\_HDF5 -DHAS\_OPENBLAS -DNDEBUG -D\_SCL\_SECURE\_NO\_WARNINGS -DUSE\_OPENCV -DUSE\_LEVELDB -DUSE\_LMDB -DWITH\_PYTHON\_LAYER -DBOOST\_PYTHON\_STATIC\_LIB -DUSE\_CUDNN -D\_UNICODE -DUNICODE -Xcompiler "/EHsc /W1 /nologo /Ox /FS /Zi  /MD " -o C:\zhh\caffe-master\windows\..\Build\Int\libcaffe\x64\Release\roi\_pooling\_layer.cu.obj "C:\zhh\caffe-master\src\caffe\layers\roi\_pooling\_layer.cu"”已退出，返回代码为 2。

1>C:\zhh\caffe-master\include\caffe/util/cudnn.hpp(114): error : too few arguments in function call

解决方法：

C:\zhh\caffe-master\include\caffe\util\cudnn.hpp, 114行改为：

#if CUDNN\_VERSION\_MIN(6, 0, 0)

  CUDNN\_CHECK(cudnnSetConvolution2dDescriptor(\*conv,

      pad\_h, pad\_w, stride\_h, stride\_w, 1, 1, CUDNN\_CROSS\_CORRELATION,

      dataType<Dtype>::type));

#else

      CUDNN\_CHECK(cudnnSetConvolution2dDescriptor(\*conv,

      pad\_h, pad\_w, stride\_h, stride\_w, 1, 1, CUDNN\_CROSS\_CORRELATION));

#endif

（4）VS  Build libCaffe报错：

1>C:\zhh\NugetPackages\glog.0.3.3.0\build\native\glog.targets(346,5): error MSB4062: 未能从程序集 C:\zhh\NugetPackages\gflags.2.1.2.1\build\native\\private\coapp.NuGetNativeMSBuildTasks.dll 加载任务“NuGetPackageOverlay”。未能加载文件或程序集“file:///C:\zhh\NugetPackages\gflags.2.1.2.1\build\native\private\coapp.NuGetNativeMSBuildTasks.dll”或它的某一个依赖项。系统找不到指定的文件。 请确认 <UsingTask> 声明正确，该程序集及其所有依赖项都可用，并且该任务包含实现 Microsoft.Build.Framework.ITask 的公共类。

1>C:\zhh\NugetPackages\glog.0.3.3.0\build\native\glog.targets(346,5): error MSB4062: 未能从程序集 C:\zhh\NugetPackages\gflags.2.1.2.1\build\native\\private\coapp.NuGetNativeMSBuildTasks.dll 加载任务“NuGetPackageOverlay”。未能加载文件或程序集“file:///C:\zhh\NugetPackages\gflags.2.1.2.1\build\native\private\coapp.NuGetNativeMSBuildTasks.dll”或它的某一个依赖项。系统找不到指定的文件。 请确认 <UsingTask> 声明正确，该程序集及其所有依赖项都可用，并且该任务包含实现 Microsoft.Build.Framework.ITask 的公共类。

解决方法：

这个是重新build时经常遇到的问题，删除C:\zhh\NugetPackages\gflags.2.1.2.1，然后重新编译即可

（5）pyCharm中运行Faster RCNN的train\_net.py报错：

 [0.1 0.1 0.2 0.2]

 [0.1 0.1 0.2 0.2]

 [0.1 0.1 0.2 0.2]

 [0.1 0.1 0.2 0.2]]

[0.1 0.1 0.2 0.2]

Normalizing targets

done

\*\*\* Check failure stack trace: \*\*\*

检查发现是train.py中以下语句报错

self.solver = caffe.SGDSolver(solver\_prototxt)

解决方法：

猜测是solver\_prototxt的路径问题，pycharm——Run——Edit Configurations, 将工作路径由C:\zhh\py-faster-rcnn\tools改为C:\zhh\py-faster-rcnn，并将参数中的路径改为相对路径，如：

--gpu 0 --solver .\models\pascal\_voc\VGG16\faster\_rcnn\_end2end\solver.prototxt --imdb voc\_2007\_trainval --iters 100000 --weights .\output\faster\_rcnn\_end2end\voc\_2007\_trainval\vgg16\_faster\_rcnn\_iter\_30000\_1.caffemodel --cfg .\experiments\cfgs\faster\_rcnn\_end2end.yml

（6）运行train\_net.py报错：

  File "C:\zhh\py-faster-rcnn\tools\..\lib\roi\_data\_layer\layer.py", line 87, in setup

    layer\_params = yaml.load(self.param\_str\_)

AttributeError: 'RoIDataLayer' object has no attribute 'param\_str\_'

解决方法：

用Notepad++将所有.py文件中的self.param\_str\_都改为self.param\_str

（7）运行train\_net.py报错：

Traceback (most recent call last):

  File "C:/zhh/py-faster-rcnn/tools/train\_net.py", line 112, in <module>

    max\_iters=args.max\_iters)

  File "C:\zhh\py-faster-rcnn\tools\..\lib\fast\_rcnn\train.py", line 159, in train\_net

    model\_paths = sw.train\_model(max\_iters)

  File "C:\zhh\py-faster-rcnn\tools\..\lib\fast\_rcnn\train.py", line 100, in train\_model

    self.solver.step(1)

  File "C:\zhh\py-faster-rcnn\tools\..\lib\rpn\proposal\_target\_layer.py", line 66, in forward

    rois\_per\_image, self.\_num\_classes)

  File "C:\zhh\py-faster-rcnn\tools\..\lib\rpn\proposal\_target\_layer.py", line 191, in \_sample\_rois

    \_get\_bbox\_regression\_labels(bbox\_target\_data, num\_classes)

  File "C:\zhh\py-faster-rcnn\tools\..\lib\rpn\proposal\_target\_layer.py", line 127, in \_get\_bbox\_regression\_labels

    bbox\_targets[ind, start:end] = bbox\_target\_data[ind, 1:]

TypeError: slice indices must be integers or None or have an \_\_index\_\_ method

解决方法：

C:\zhh\py-faster-rcnn\lib\rpn\proposal\_target\_layer.py 第60行改为

fg\_rois\_per\_image = np.round(cfg.TRAIN.FG\_FRACTION \* rois\_per\_image).astype(int)

第125行，126行的：

        start = 4 \* cls

        end = start + 4

改为：

        start = int(4 \* cls)

        end = int(start + 4)

（8） lib目录下setup安装方法

lib目录下运行

python setup.py install

python setup.py build\_ext --inplace（这一条语句生成cpu版本的pyd文件）

python setup\_cuda.py install

python setup\_cuda.py build\_ext --inplace（这一条语句生成gpu版本的pyd文件）

1. Windows下编译及调试Debug版Caffe

Caffe 版本：<https://github.com/Microsoft/caffe>

前期Caffe安装及Release版编译步骤参考：

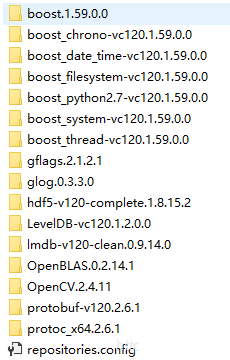
<https://blog.csdn.net/AManFromEarth/article/details/80212554>

<https://blog.csdn.net/zh8706/article/details/94553421>

（1）检查工程所缺依赖文件

下面要在VS2013下编译Debug版Caffe。

我们注意到在Release模式下编译Caffe，会在caffe-master所在的目录下创建NugetPackages目录，包含了编译Release版Caffe所需要的工具包，如下：



在VS2013中切换为Debug x64模式，编译libCaffe或者Caffe工程，会报"error MSB4062: 未能从程序集..."及"error : NuGet Error:未知命令:“overlay”的错误，如下：



在这里需要特别说明下，在Release模式下编译时可能也会出现这个错误，这个问题的原因是Caffe编译过程中VS的NuGet程序包管理工具会下载工程所需要的程序包文件，但有时候却下载不全，就出现这个错误。所以在Release模式下，如果都编译成功了，就表示所依赖的工具包是完整下载的，没有缺漏，所以最好备份一下整个NugetPackages文件夹。

但是即使我们在Release模式下编译成功，切换到Debug模式下依然还是会报以上错误的，原因就是NuGet无法下载Debug版本的工具包，这里所缺的工具包有两个：glog.0.3.3.0与OpenCV.2.4.11的Debug版本dll及lib文件

NugetPackages\glog.0.3.3.0\build\native\bin\x64\v120目录下缺Debug版本dll文件

NugetPackages\glog.0.3.3.0\build\native\lib\x64\v120目录下缺Debug版本lib文件

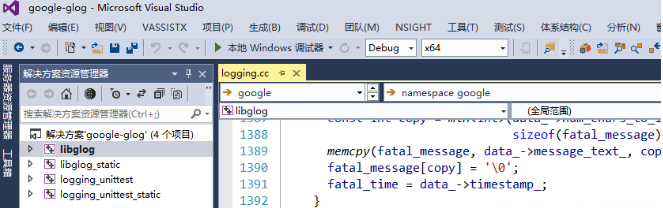
NugetPackages\OpenCV.2.4.11\build\native\bin\x64\v120目录下缺Debug版本dll文件

NugetPackages\OpenCV.2.4.11\build\native\lib\x64\v120目录下缺Debug版本lib文件

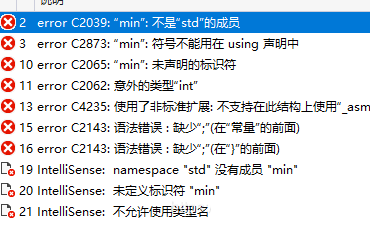
既然NuGet没法帮我们下载（源端估计也没有提供）Debug版本文件，那我们只能自己下载。

（2）编译生成glog

到这里<https://github.com/google/glog/releases/tag/v0.3.3> 下载glog0.3.3，下载后使用VS2013打开google-glog.sln文件，切换到Debug x64平台。

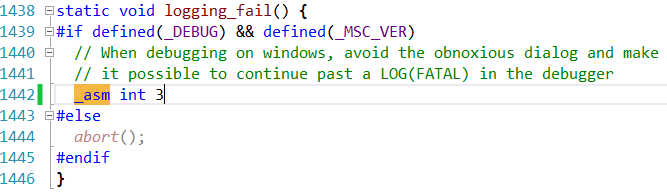


对libglog右键--生成，会报以下错误：



这里有两个错误，一个是error C2039: “min”: 不是“std”的，解决方法是：在logging.cc中添加头文件，即加上 #include <algorithm>

还有一个错误是logging\_fail函数中用到了汇编代码，但在x64模式下，VS是不支持汇编代码的。



在这里我们用最简单的解决方式，即将\_asm int 3一行 注释掉即可，有兴趣的朋友也可以查查\_asm int 3有什么作用。

然后再重新生成libglog，即生成成功。

将glog-0.3.3\x64\Debug下的dll文件拷贝到到NugetPackages\glog.0.3.3.0\build\native\bin\x64\v120\Debug\dynamic

将glog-0.3.3\x64\Debug下的lib文件拷贝到NugetPackages\glog.0.3.3.0\build\native\lib\x64\v120\Debug\dynamic

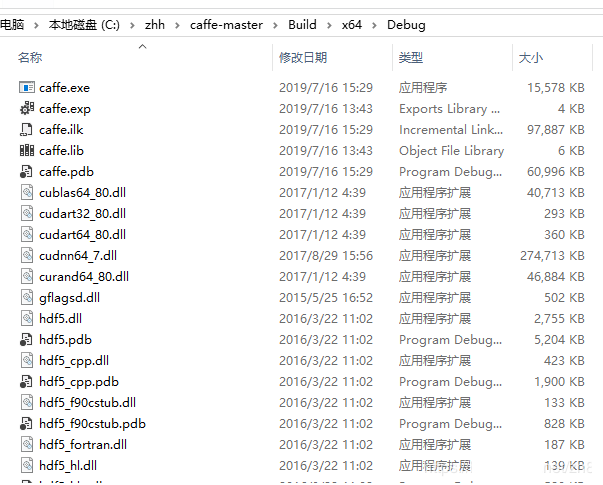
（3）下载OpenCV 2.4.11

到这里https://opencv.org/releases/page/4/下载OpenCV2.4.11.exe，下载后双击运行解压，

将opencv\build\x64\vc12\bin目录下的\*d.dll结尾的dll文件全部拷贝到NugetPackages\OpenCV.2.4.11\build\native\bin\x64\v120\Debug

将opencv\build\x64\vc12\lib目录下的\*d.lib结尾的lib文件全部拷贝到NugetPackages\OpenCV.2.4.11\build\native\lib\x64\v120\Debug

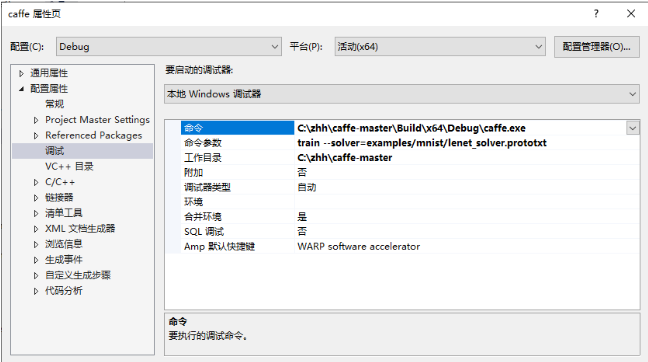
至此Debug x64模式下Caffe所缺的文件已经齐全，再次编译libcaffe及caffe均可编译成功，如下：



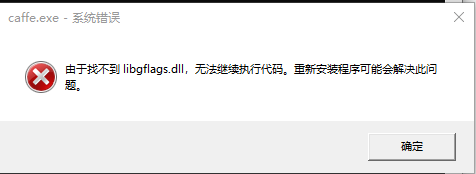
 （4）调试Caffe

下面我们以caffe中的mnist样例为例调试caffe。

在VS中将caffe设为启动项目，并右键——属性——调试，设置如下：



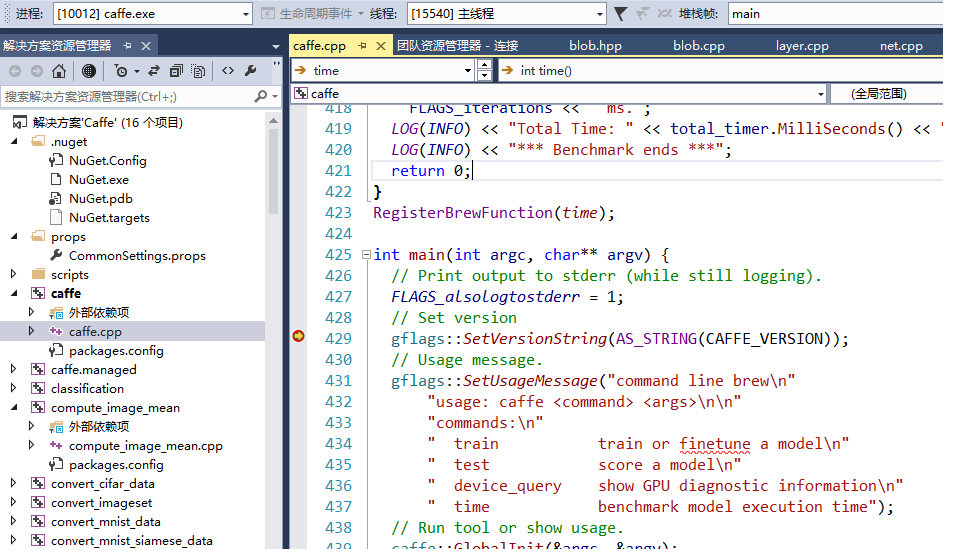
点击VS工具栏中的https://img-blog.csdnimg.cn/2019071616024463.png开始调试，但弹出了以下错误：



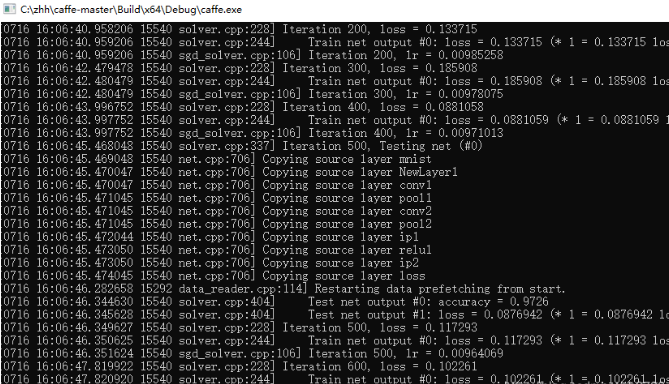
 解决方法：

找到caffe-master\Build\x64\Debug目录下的libgflags-debug.dll文件，将其重命名为libgflags.dll即可。

再次调试，成功命中caffe入口main函数断点：

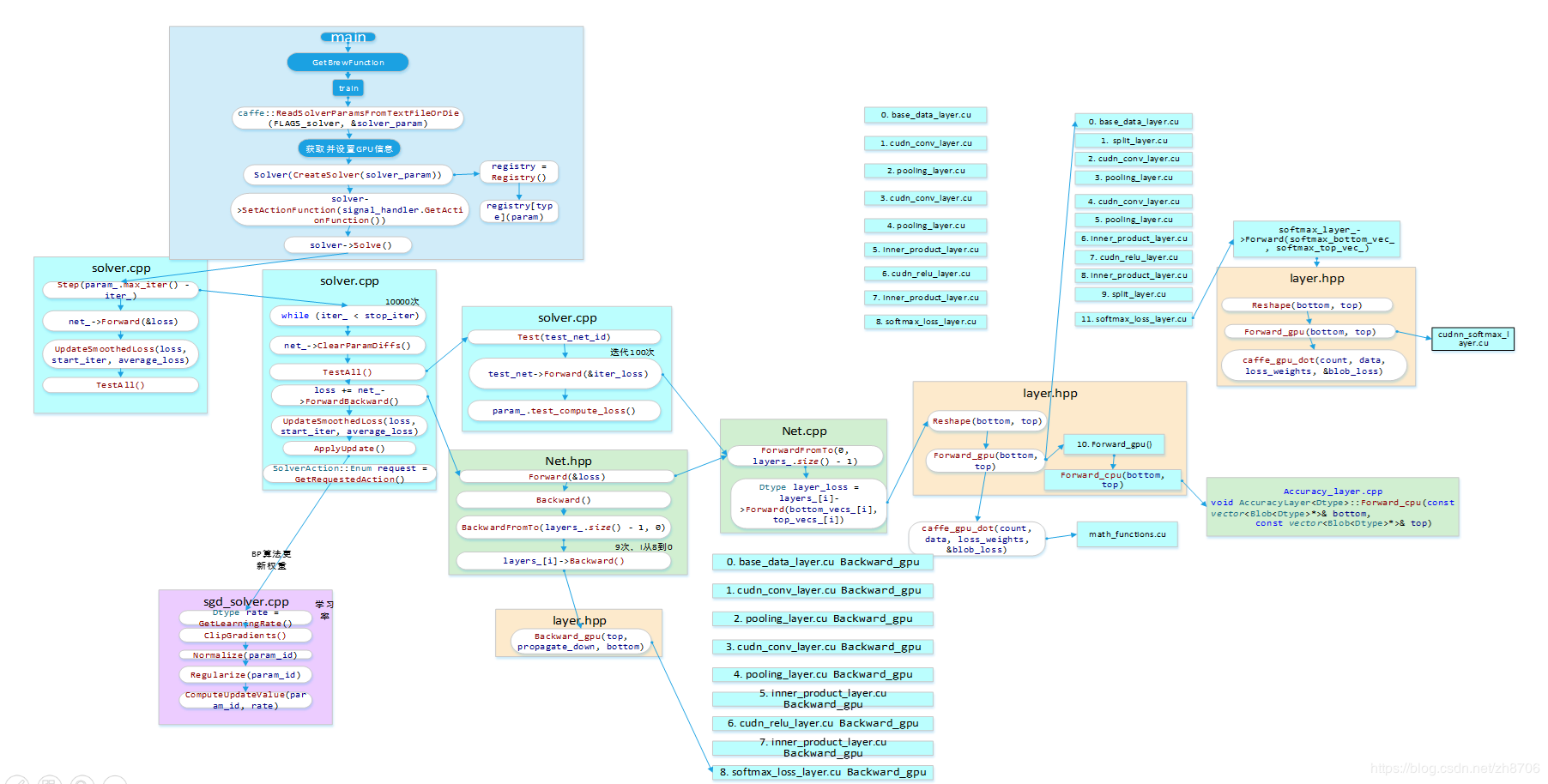


也出现了mnist训练的过程记录日志：



至此，caffe debug版本编译及调试完成。

1. Caffe源码函数调用关系图



1. Caffe中Solver、Net、Layer、Blob的构建流程

以训练流程的构建为例：

首先在**caffe.cpp**的**train()**中调用caffe::**SolverRegistry<float>::CreateSolver(solver\_param)**：

|  |
| --- |
| shared\_ptr<caffe::Solver<float> >  solver(caffe::SolverRegistry<float>::CreateSolver(solver\_param)); |

即通过solover\_factory.hpp中的工厂机制（registry[type](param)）new 一个具体的已注册的Solver的子类（例如new SGDSolver<Dtype>(param)）的实例。而在SGDSolver的构造函数中，首先使用SolverParameter来初始化父类Solver，如下：

|  |
| --- |
| explicit SGDSolver(const SolverParameter& param)  : Solver<Dtype>(param) { PreSolve(); } |

而在父类**Solver**的构造函数中调用**Slover::Init()，**在**Slover::Init()**中又调用**Slover::InitTrainNet()，**而在**InitTrainNet()**中首先从**SolverParameter**参数中解析出**Net**所需要的参数，

|  |
| --- |
| if (param\_.has\_net()) {  LOG\_IF(INFO, Caffe::root\_solver())  << "Creating training net from net file: " << param\_.net();  ReadNetParamsFromTextFileOrDie(param\_.net(), &net\_param);  } |

然后再通过**net\_.reset(new Net<Dtype>(net\_param))**new一个**Net**网络实例：

|  |
| --- |
| if (Caffe::root\_solver()) {  net\_.reset(new Net<Dtype>(net\_param));  } else {  net\_.reset(new Net<Dtype>(net\_param, root\_solver\_->net\_.get()));  } |

进入到**Net**类中，在**Net**的构造函数中调用**Net::Init(),**在**Net::Init()**中的for循环中根据总层数逐次调用**LayerRegistry<Dtype>::CreateLayer(layer\_param)：**

|  |
| --- |
| layers\_.push\_back(LayerRegistry<Dtype>::CreateLayer(layer\_param)); |

即通过layer\_factory.cpp中的工厂机制new 一个具体的已注册的layer层，这里是根据Net传进来的layer\_param获取到层类型再调用registry[type](param)。例如首先是DataLayer层, 使用Net中传进来的该层的参数LayerParameter初始化DataLayer层：

|  |
| --- |
| template <typename Dtype>  DataLayer<Dtype>::DataLayer(const LayerParameter& param)  : BasePrefetchingDataLayer<Dtype>(param),  reader\_(param) {  } |

然后又在Net::Init()中for循环中new 下一层，例如Convolution。Net中new出来的子层都放到layers\_队列中。

另外，每个for循环中，new出一个Layer后，又通过AppendTop()与AppendBottom()等函数将各层的bottom与top以及weight值构建出一个链路出来，bottom、top、weight都是Blob结构，这里一个top就是一个Blob，例如在mnist的Data层（对应于Caffe中的DataLayer层）中一个top: "data"（一个Blob）即表示一个批次的元素，其维度信息为（64,1,28,28）,一个批次为64张图片，长宽均为28。另外一个top: "label"，其维度信息为（64,1），这也是一个Blob。

Net构造函数中new出所有的子层后回到Slover::InitTrainNet()中，Solver用new出来的Net来初始化其成员 shared\_ptr<Net<Dtype> > net\_; InitTrainNet()构建完训练网络后，Slover::Init()又调用Solver<Dtype>::InitTestNets()使用和InitTrainNet()一样的方法构建测试网络（即训练过程中的验证操作），其成员变量为vector<shared\_ptr<Net<Dtype> > > test\_nets\_。

以上调用完成后遍回到caffe.cpp的train()中，即完成了Solver、Net、Layer的构建。以上流程说明了，在Caffe中Blob就是砖头、Layer是墙，Net是大厦，而Solver就是图纸。

1. Windows训练与测试Caffe mnist

参考以下两篇文章来做训练（任选一）：

1. windows下caffe训练mnist数据集（https://blog.csdn.net/panhong1992/article/details/84593439）

2. Windows caffe (一) MNIST手写体跑起来（https://blog.csdn.net/renyhui/article/details/60871888）

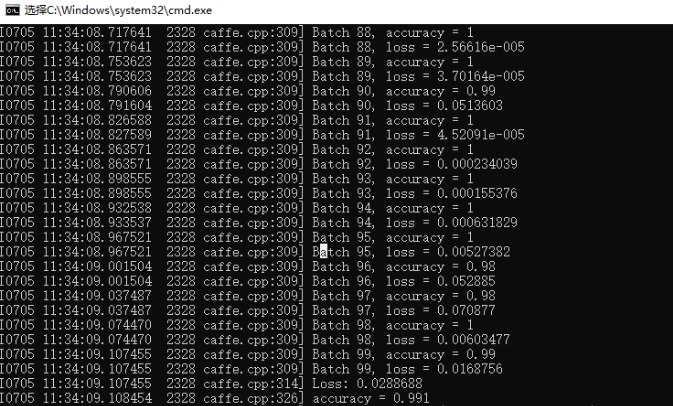
第1篇文章中只写了训练mnist，可参考第2篇文章多了测试的步骤，即：

模型训练好后，在examples\mnist\mnist\_data\ 目录下多了caffemodel模型文件，因此在caffe-master目录下创建test\_mnist.bat文件，并运行即可

|  |
| --- |
| .\Build\x64\Release\caffe.exe test -model=examples\mnist\lenet\_train\_test.prototxt -weights=examples\mnist\mnist\_data\lenet\_iter\_10000.caffemodel -iterations=100  pause |

注意这里使用到的是lenet\_train\_test.prototxt文件，因此这里的测试是对训练样本中的验证集数据进行测试。

运行结果如下：



测试单张图片

创建一张28x28, 256位黑底白字的图片，如下，命名为num.bmp, 放到examples/images/路径下

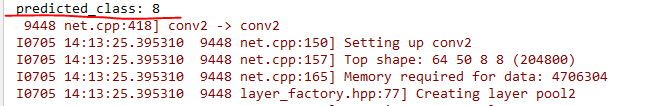


在examples目录下创建python文件test\_mnist\_num.py，代码如下：

注意这里的网络文件为lenet.prototxt，为真正的测试prototxt文件，即deploy部署文件

|  |
| --- |
| import os.path as osp  import caffe    this\_dir = osp.dirname(\_\_file\_\_)  model\_file = osp.join(this\_dir, 'mnist/lenet.prototxt')  pretrained = osp.join(this\_dir, 'mnist/mnist\_data/lenet\_iter\_10000.caffemodel')  image\_file = osp.join(this\_dir, 'images/num.bmp')    input\_image = caffe.io.load\_image(image\_file, color=False)  net = caffe.Classifier(model\_file, pretrained)  prediction = net.predict([input\_image], oversample=False)  caffe.set\_mode\_gpu()    print 'predicted\_class:', prediction[0].argmax() |

运行后，结果如下：



1. Caffe solver文件参数详细解析

以caffe-master\examples\mnist\lenet\_solver.prototxt文件来解析，这个文件为训练mnist模型时的超参数文件，这个文件中中有两个参数test\_iter与test\_interval，这里用了单词“test”，但这个超参数时在训练阶段使用，而不是测试阶段，因此我认为应该翻译为验证更合适。因此文件参数解析如下：

|  |
| --- |
| test\_iter: 100 #一次验证计算的图片批次，一次验证计算所有验证集图片（10000张），分test\_iter(100)次进行，每个批次包含64（batch\_size）张验证集图片。也就是说一次验证完成100个批次的图片验证    test\_interval: 500 #每训练500次（即迭代500次），进行一次验证。    base\_lr: 0.01 #基础学习率，即初始训练时的学习率  momentum: 0.9 #梯度下降法中的参数，一般不需变化  weight\_decay: 0.0005 #权重衰减项，即正则化项前面的系数，用于防止过拟合    lr\_policy: "inv" #训练过程中学习率调整的方法，如果设置为inv, 学习率更新为 base\_lr \* (1 + gamma \* iter) ^ (- power)  gamma: 0.0001 #学习率变化的比率  power: 0.75 #inv中的参数    display: 100 #每训练（迭代）100次显示一次训练的结果    max\_iter: 10000 #总共训练（迭代）10000次    snapshot: 5000 #每5000次训练（迭代）输出caffemodel权重文件  snapshot\_prefix: "examples/mnist/mnist\_data/lenet" #权重文件路径    solver\_mode: GPU #使用GPU训练 |

训练过程中的参数对应如下：





1. Windows使用Cifar10训练及测试Caffe版DenseNet

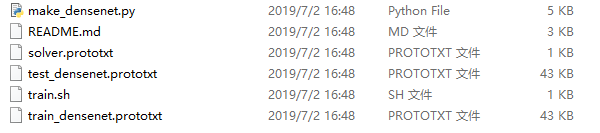
论文名称：Densely Connected Convolutional Networks（CVPR 2017, Best Paper Award）

论文链接：<https://arxiv.org/pdf/1608.06993.pdf>

源码链接：<https://github.com/liuzhuang13/DenseNet>

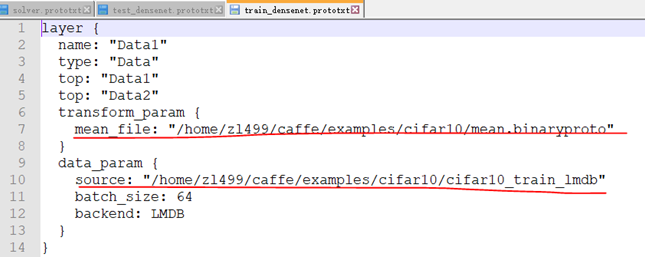
caffe版源码： <https://github.com/liuzhuang13/DenseNetCaffe>

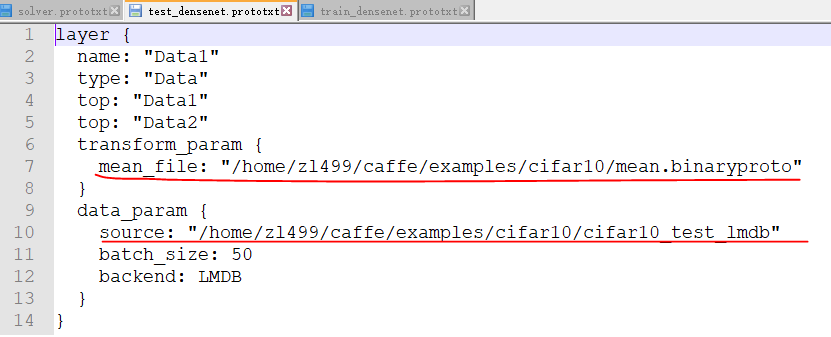
（1）下载Caffe版DenseNet源码

源码中包含以下6个文件：  


|  |  |
| --- | --- |
| make\_densenet.py | 用于生成train\_densenet.prototxt文件、test\_densenet.prototxt文件以及solver.prototxt文件 |
| train\_densenet.protoxt | 训练时用到的模型网络文件 |
| test\_denset.prototxt | 训练过程中验证部分用到的网络文件 |
| solover.prototxt | caffe用到的solover文件 |
| train.sh | ubuntu环境下的训练文件 |

train\_densenet.prototxt与test\_densenet.prototxt文件指定了需要用到cifar10数据集及其均值，如图。





因此我们需要先准备cifar10数据集。

（2）创建cifar10数据集

2. 1 下载cifar10数据集

官网下载bin格式的cifar10  ： http://www.cs.toronto.edu/~kriz/cifar.html

即CIFAR-10 binary version (suitable for C programs)格式版本，解压存放在 ：caffe-master\examples\cifar10\input\_folder当中（input\_folder文件夹需要自己创建），

如图：



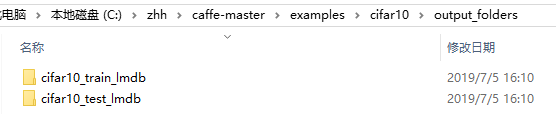
2. 2 转换为lmdb格式

首先在VS中编译出Caffe的convert\_cifar\_data.exe工具，在caffe-master\Build\x64\Release\目录下。

在caffe-master\examples\cifar10 下创建一个bat文件(convert\_cifar10\_lmdb.bat)，输入以下:

|  |
| --- |
| ..\..\Build\x64\Release\convert\_cifar\_data.exe input\_folder output\_folders lmdb  pause |

然后双击运行，可以看到在 caffe-master\examples\cifar10下会生成output\_folders文件夹，里面存放的就是转换好lmdb格式数据，如图：



如果需要leveldb格式，则将bat文件中的lmdb改为lveldb，再运行即可。

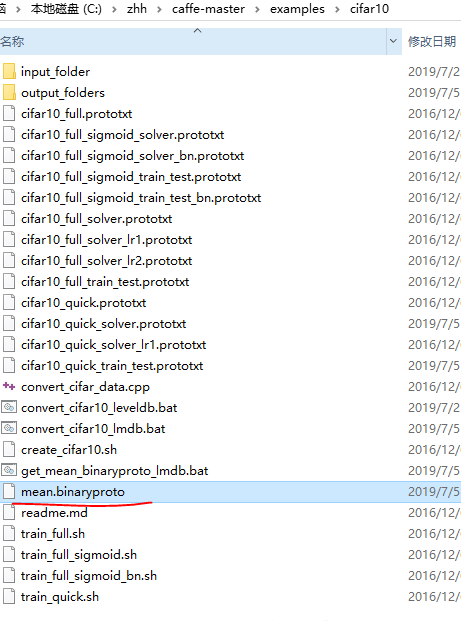
2.3 生成mean.binaryproto均值文件

首先在VS中编译出Caffe的compute\_image\_mean.exe工具，在caffe-master\Build\x64\Release\目录下。

在caffe-master\examples\cifar10 下创建一个bat文件(get\_mean\_binaryproto\_lmdb.bat)，输入以下:

|  |
| --- |
| ..\..\Build\x64\Release\compute\_image\_mean.exe -backend=lmdb ../../examples\cifar10\output\_folders\cifar10\_train\_lmdb mean.binaryproto  Pause |

双击运行后，即在affe-master\examples\cifar10下生成lmdb格式的mean.binaryproto文件，如图：

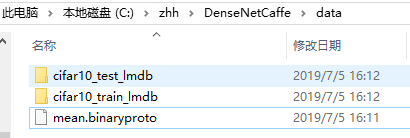


同理如果需要leveldb格式，修改bat文件中的lmdb为leveldb即可。

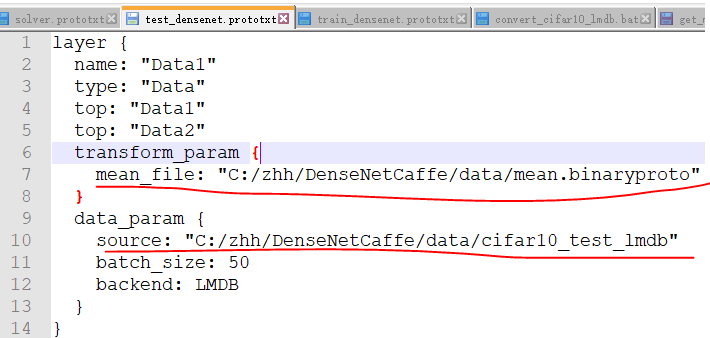
（3）训练Cifar10

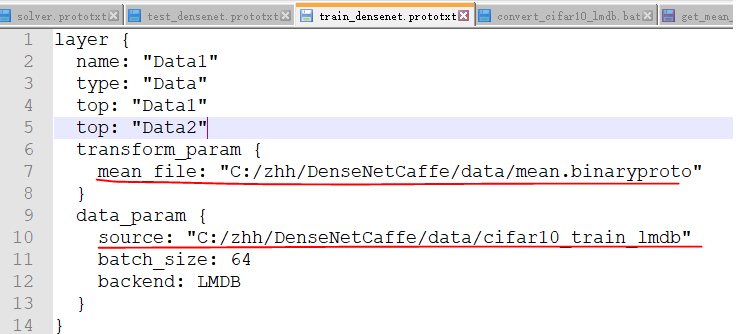
3.1 更改模型文件数据集路径

在DenseNetCaffe根目录下创建data文件夹，将cifar10\_train\_lmdb与cifar10\_test\_lmdb两个文件夹以及mean.binaryproto文件到拷到data下，如图：

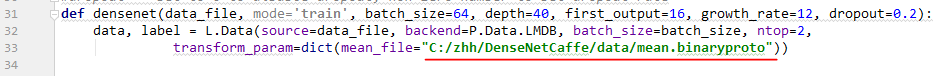


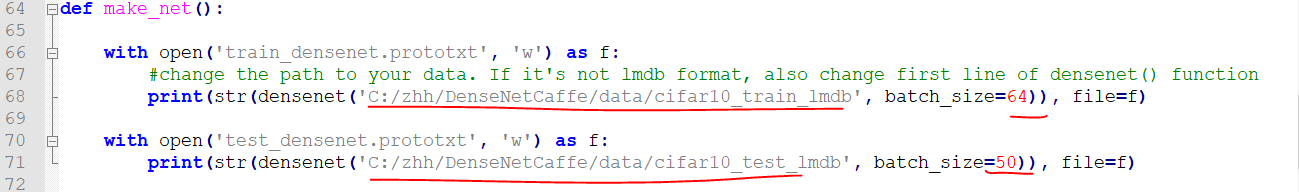
更改train\_densenet.prototxt与test\_densenet.prototxt文件中的数据集路径即均值文件路径





也可以更改make\_densenet.py文件中的路径来生成3个prototxt文件，如图：



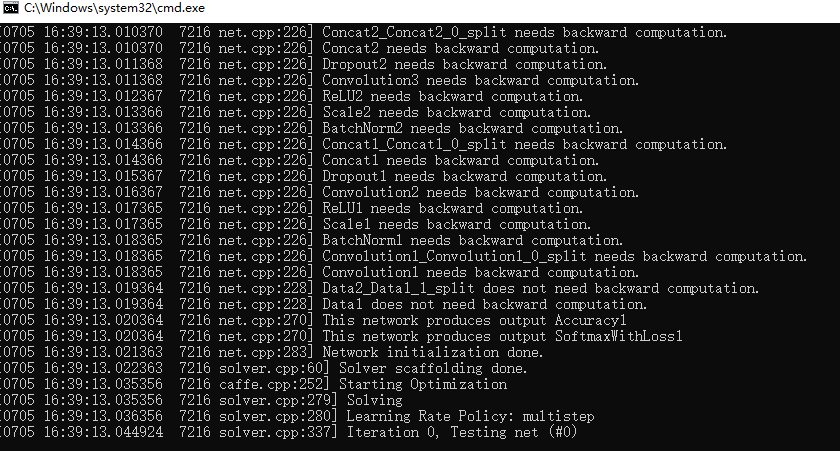


3.2 训练模型

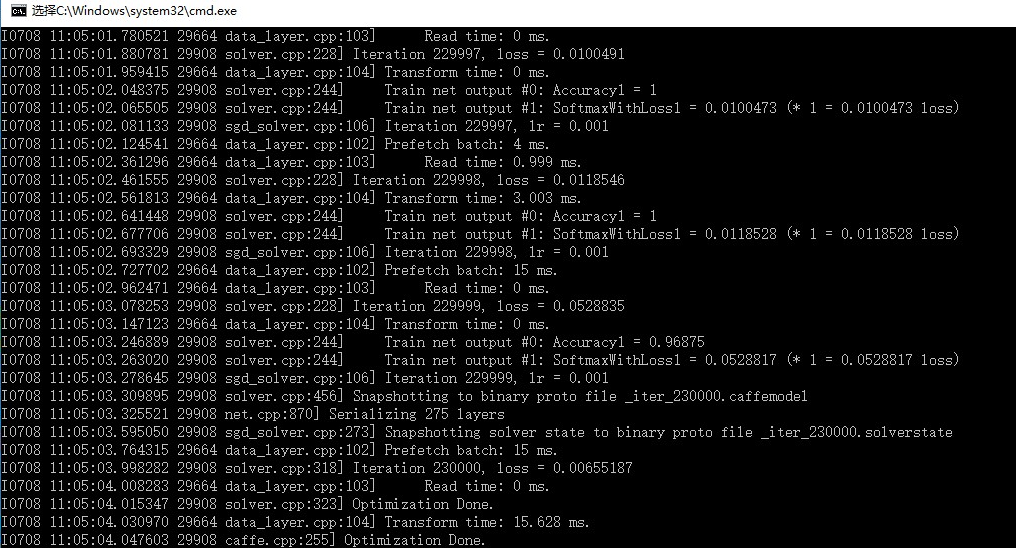
在DenseNetCaffe根目录下创建一个train.bat文件，编辑以下代码：

|  |
| --- |
| SET GLOG\_logtostderr=1  C:\zhh\caffe-master\Build\x64\Release\caffe.exe train --solver C:\zhh\DenseNetCaffe\solver.prototxt  pause |

另外，需要注意的是，原模型训练需要较大的显存，我本机显存为8GB，运行train.bat文件加载模型开始训练后会报out of memory错误，即显存不足。可通过降低train\_densenet.prototxt文件中的batch\_size来降低所需要的显存，这里我将batch\_size修改为32，运行train.bat，即开始训练，如图：



迭代次数达到设定的最高值23万时，模型训练停止，模型训练的准确率已接近1，损失值接近0。



在DensetNet根目录下生成了权重文件，如下：

https://img-blog.csdnimg.cn/20190708214055274.png

（4）测试模型

4.1 对测试集图像数据进行测试

在DensetNet根目录下创建test.bat，如下：

|  |
| --- |
| SET GLOG\_logtostderr=1  C:\zhh\caffe-master\Build\x64\Release\caffe.exe test -model=C:\zhh\DenseNetCaffe\test\_densenet.prototxt -weights=C:\zhh\DenseNetCaffe\\_iter\_230000.caffemodel -iterations=100  pause |

双击运行，得到对100个批次测试集数据的测试，如下：

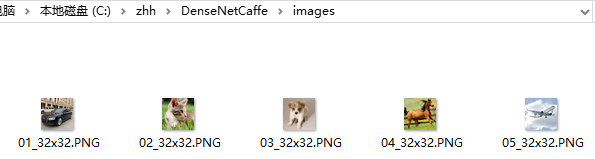
|  |
| --- |
| I0708 13:51:39.630722 14688 caffe.cpp:286] Running for 100 iterations.  I0708 13:51:43.784732 14688 caffe.cpp:309] Batch 0, Accuracy1 = 0.96  I0708 13:51:44.222826 14688 caffe.cpp:309] Batch 0, SoftmaxWithLoss1 = 0.0977751  I0708 13:51:47.511251 14688 caffe.cpp:309] Batch 1, Accuracy1 = 0.92  I0708 13:51:47.512169 14688 caffe.cpp:309] Batch 1, SoftmaxWithLoss1 = 0.485769  I0708 13:51:50.807596 14688 caffe.cpp:309] Batch 2, Accuracy1 = 0.9  I0708 13:51:50.807596 14688 caffe.cpp:309] Batch 2, SoftmaxWithLoss1 = 0.317953  I0708 13:51:54.090023 14688 caffe.cpp:309] Batch 3, Accuracy1 = 0.94  I0708 13:51:54.090023 14688 caffe.cpp:309] Batch 3, SoftmaxWithLoss1 = 0.216393  I0708 13:51:57.379446 14688 caffe.cpp:309] Batch 4, Accuracy1 = 0.94  I0708 13:51:57.379446 14688 caffe.cpp:309] Batch 4, SoftmaxWithLoss1 = 0.395111  I0708 13:52:00.693857 14688 caffe.cpp:309] Batch 5, Accuracy1 = 0.92  I0708 13:52:00.693857 14688 caffe.cpp:309] Batch 5, SoftmaxWithLoss1 = 0.424328  ……………  I0708 14:00:43.962534 14688 caffe.cpp:309] Batch 95, Accuracy1 = 0.84  I0708 14:00:43.962534 14688 caffe.cpp:309] Batch 95, SoftmaxWithLoss1 = 0.692951  I0708 14:00:47.326921 14688 caffe.cpp:309] Batch 96, Accuracy1 = 1  I0708 14:00:47.327920 14688 caffe.cpp:309] Batch 96, SoftmaxWithLoss1 = 0.00866098  I0708 14:00:50.655325 14688 caffe.cpp:309] Batch 97, Accuracy1 = 1  I0708 14:00:50.656323 14688 caffe.cpp:309] Batch 97, SoftmaxWithLoss1 = 0.021941  I0708 14:00:54.095676 14688 caffe.cpp:309] Batch 98, Accuracy1 = 0.88  I0708 14:00:54.095676 14688 caffe.cpp:309] Batch 98, SoftmaxWithLoss1 = 0.576406  I0708 14:00:57.427078 14688 caffe.cpp:309] Batch 99, Accuracy1 = 0.9  I0708 14:00:57.427078 14688 caffe.cpp:309] Batch 99, SoftmaxWithLoss1 = 0.586738  I0708 14:00:57.428077 14688 caffe.cpp:314] Loss: 0.312385  I0708 14:00:57.429077 14688 caffe.cpp:326] Accuracy1 = 0.9232  I0708 14:00:57.429077 14688 caffe.cpp:326] SoftmaxWithLoss1 = 0.312385 (\* 1 = 0.312385 loss) |

4.2 测试其他图片

这里我们编写Python文件调用Caffe对我们自己的图片文件进行测试。

4.2.1 准备测试图片

在densenet根目录下创建images文件夹，存放要测试的图片，如下，图像为RGB格式，32x32大小



4.2.2生成densenet.prototxt文件

4.1中使用的prototxt文件为test\_densenet.prototxt，为训练时用到的网络文件，实际在测试及部署需要deploy prototxt文件，可在test\_densenet.prototxt上改写，如下步骤：

拷贝test\_densenet.prototxt文件，并重命名为densent.prototxt

在densent.prototxt中，删除第1层，即name为Data1的层，并添加如下层：

|  |
| --- |
| layer {  name: "data"  type: "Input"  top: "data"  input\_param { shape: { dim: 1 dim: 3 dim: 32 dim: 32 } }  } |

将第2层（即nama为Convolution1的层）中的bottom: "Data1"改为bottom: "data"

删除最后一层，即name为Accuracy1的层

对最后name为SoftmaxWithLoss1的层改写，如下：

改写前：

|  |
| --- |
| layer {  name: "SoftmaxWithLoss1"  type: "SoftmaxWithLoss"  bottom: "InnerProduct1"  bottom: "Data2"  top: "SoftmaxWithLoss1"  } |

改写后：

|  |
| --- |
| layer {  name: "prob"  type: "Softmax"  bottom: "InnerProduct1"  top: "softmax"  } |

4.2.3生成mean.npy文件

Caffe中需要的是mean.binaryproto二进制文件，但Caffe的Python接口需要npy格式的文件，因此需要将mena.binaryproto转为mean.npy格式。在mean.binaryproto的目录下（DenseNetCaffe\data）创建make\_meannpy.py文件，代码如下：

|  |
| --- |
| import numpy as np  import caffe    MEAN\_PROTO\_PATH = '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) |

运行，即在data目录下生成mean.npy文件

4.2.4编写test\_densenet.py文件

在densenet根目录下创建test\_densenet.py文件，编写代码如下：

|  |
| --- |
| import os.path as osp  import numpy as np  import caffe    this\_dir = osp.dirname(\_\_file\_\_)  net\_file = osp.join(this\_dir, 'densenet.prototxt')  pretrained\_file = osp.join(this\_dir, '\_iter\_230000.caffemodel')  image\_file = osp.join(this\_dir, 'images/05\_32x32.png')  mean\_file = osp.join(this\_dir, 'data/mean.npy')    classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']    input\_image = caffe.io.load\_image(image\_file, color=True) #load\_image读进来的图像数据为RGB格式，范围为0~1    mean\_npy = np.load(mean\_file)  mean = mean\_npy.mean(1).mean(1)    caffe.set\_mode\_gpu()  net = caffe.Classifier(net\_file,  pretrained\_file,  mean=mean,  raw\_scale=255, #数据范围调整为0~255  channel\_swap=(2,1,0)) #转换为BGR格式  prediction = net.predict([input\_image], oversample=True)    idx = prediction[0].argmax()  print image\_file + ' predicted\_class:', classes[idx] |

运行以上程序，即可预测出图片的类别，如下：

https://img-blog.csdnimg.cn/20190708221258909.png

https://img-blog.csdnimg.cn/20190708221333355.png

https://img-blog.csdnimg.cn/20190708221421523.png

https://img-blog.csdnimg.cn/20190708221455605.png

https://img-blog.csdnimg.cn/20190708221225666.png

1. Caffe py-Faster-RCNN 源码解析

源码：<https://github.com/rbgirshick/py-faster-rcnn>

首先来看\tools\train\_net.py，首先parse\_args()读入命令行设置的参数信息，然后读取yml设置的参数信息，如下

代码段1：

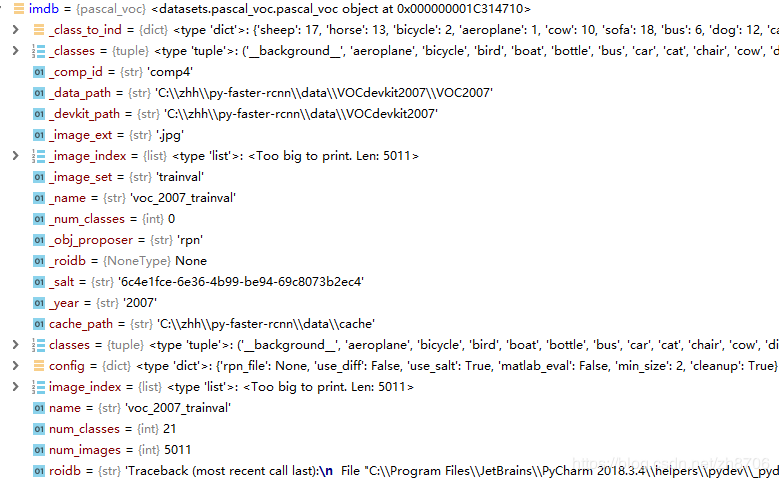
|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  args = parse\_args()    print('Called with args:')  print(args)    if args.cfg\_file is not None:  cfg\_from\_file(args.cfg\_file) #读取'.\\experiments\\cfgs\\faster\_rcnn\_end2end.yml'文件中的参数  if args.set\_cfgs is not None:  cfg\_from\_list(args.set\_cfgs)    cfg.GPU\_ID = args.gpu\_id #设置gpu id号    print('Using config:')  pprint.pprint(cfg)    if not args.randomize:  # fix the random seeds (numpy and caffe) for reproducibility  np.random.seed(cfg.RNG\_SEED)  caffe.set\_random\_seed(cfg.RNG\_SEED)    # set up caffe  caffe.set\_mode\_gpu()  caffe.set\_device(args.gpu\_id)    # imdb\_name为voc\_2007\_trainval，返回数据集，及数据集中的框标记信息（包括框类别，图片路径，宽高等），这里roidb为list，长度10022  imdb, roidb = combined\_roidb(args.imdb\_name)  print '{:d} roidb entries'.format(len(roidb))    output\_dir = get\_output\_dir(imdb)  print 'Output will be saved to `{:s}`'.format(output\_dir)    train\_net(args.solver, roidb, output\_dir,  pretrained\_model=args.pretrained\_model,  max\_iters=args.max\_iters) |

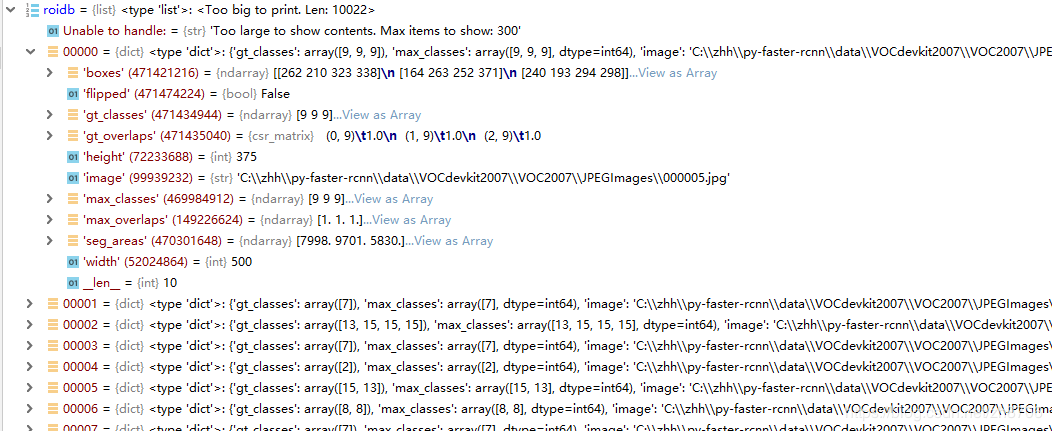
（1） combined\_roidb函数解析

代码段1中的重点函数：

|  |
| --- |
| imdb, roidb = combined\_roidb(args.imdb\_name) |

这里args.imdb\_name为voc\_2007\_trainval，这个函数根据这个数据集名称，加载数据集接口给到imdb, 而roidb则为数据集中图片的标注框坐标及类别信息（包括框类别、坐标、图片路径、图片宽高等）。先看看这个函数的输出结果：





进入该函数的实现，也是在train\_net.py文件中，如下代码段2：

代码段2：

|  |
| --- |
| def combined\_roidb(imdb\_names):#imdb\_names为数据集名称列表，'imdb\_names:voc\_2007\_trainval'  def get\_roidb(imdb\_name):#imdb\_names:voc\_2007\_trainval  imdb = get\_imdb(imdb\_name) #返回voc2007数据集  print 'Loaded dataset `{:s}` for training'.format(imdb.name)  imdb.set\_proposal\_method(cfg.TRAIN.PROPOSAL\_METHOD) #PROPOSAL\_METHOD:'gt', 设置调用的roi函数，在pascal\_voc中的gt\_roidb  print 'Set proposal method: {:s}'.format(cfg.TRAIN.PROPOSAL\_METHOD)  roidb = get\_training\_roidb(imdb) #翻转样本（可选），并获取数据集的图片路径、图片宽高、框的类别等，返回这些信息imdb.roidb  return roidb    roidbs = [get\_roidb(s) for s in imdb\_names.split('+')] #获取各个数据集的训练样本的标注框信息  roidb = roidbs[0] #获取第一个数据集的标注框信息  if len(roidbs) > 1: #如果是多个数据集  for r in roidbs[1:]: #将其他数据集的数据添加到roidb末尾  roidb.extend(r)  imdb = datasets.imdb.imdb(imdb\_names)  else:  imdb = get\_imdb(imdb\_names)  return imdb, roidb |

代码段2的第10行，imdb\_names是一个列表，数据集名称用‘+’分割，这里调用get\_roidb获取各个数据集的信息。在get\_roidb中，调用lib\datasets\factory.py中的get\_imdb获取数据集索引，如下代码段3，\_\_sets在lib\datasets\pascal\_voc.py中已加载voc2007数据集的信息

代码段3：

|  |
| --- |
| **def get\_imdb(name):**  **"""Get an imdb (image database) by name."""**  **if not \_\_sets.has\_key(name):**  **raise KeyError('Unknown dataset: {}'.format(name))**  **return \_\_sets[name]() #返回已加载的voc2007数据集** |

代码段2的第5行，imdb.set\_proposal\_method(cfg.TRAIN.PROPOSAL\_METHOD) 设置训练样本标注框的获取方式，这里PROPOSAL\_METHOD为‘gt’，调用到lib\datasets\imdb.py以下函数，

代码段4：

|  |
| --- |
| **def set\_proposal\_method(self, method):#method:'gt'**  **method = eval('self.' + method + '\_roidb') #self.gt\_roidb,在子类pascal\_voc中实现**  **self.roidb\_handler = method #将self.roidb\_handler绑定为method函数接口** |

method为self.gt\_roidb，这里将imdb.py文件中的imdb类本身并没有gt\_roidb函数，该函数的实现由pascal\_voc.py中的pascal\_voc类来实现，而pascal\_voc继承了imdb类，gt\_roidb的实现如下：

代码段5：

|  |
| --- |
| def gt\_roidb(self):  """  Return the database of ground-truth regions of interest.  This function loads/saves from/to a cache file to speed up future calls.  """  #cache\_path:'C:\\zhh\\py-faster-rcnn\\data\\cache', 'voc\_2007\_trainval'  cache\_file = os.path.join(self.cache\_path, self.name + '\_gt\_roidb.pkl')#'C:\\zhh\\py-faster-rcnn\\data\\cache\\voc\_2007\_trainval\_gt\_roidb.pkl'  if os.path.exists(cache\_file):  with open(cache\_file, 'rb') as fid:  roidb = cPickle.load(fid)  print '{} gt roidb loaded from {}'.format(self.name, cache\_file)  return roidb    gt\_roidb = [self.\_load\_pascal\_annotation(index)  for index in self.image\_index]  with open(cache\_file, 'wb') as fid:  cPickle.dump(gt\_roidb, fid, cPickle.HIGHEST\_PROTOCOL)  print 'wrote gt roidb to {}'.format(cache\_file)    return gt\_roidb |

gt\_roidb就是将py-faster-rcnn\\data\\cache\\voc\_2007\_trainval\_gt\_roidb.pkl中的标注框信息解析出来。

在回来看代码段2中的第7行，  roidb = get\_training\_roidb(imdb) ，这个函数从voc2007数据集中获取标注信息，roidb为一个List，大小为10022，在get\_trainning\_roidb函数中，为了扩充数据，对voc2007的5011张训练图片做了翻转处理，得到10022张训练集图片，roidb的信息如下：



再来看get\_trainning\_roidb的实现，这个函数在\tools\train.py中实现如下：

代码段6：

|  |
| --- |
| #翻转样本（可选），并获取数据集的图片路径、图片宽高、框的类别等，返回这些信息imdb.roidb  def get\_training\_roidb(imdb):  """Returns a roidb (Region of Interest database) for use in training."""  if cfg.TRAIN.USE\_FLIPPED: #是否翻转样本，这里是True  print 'Appending horizontally-flipped training examples...'  imdb.append\_flipped\_images() #对样本做左右翻转，样本数量翻倍  print 'done'    print 'Preparing training data...'  rdl\_roidb.prepare\_roidb(imdb) #获取数据集的图片路径、图片宽高、框的类别等  print 'done'    return imdb.roidb |

**代码段6**中的第6行imdb.append\_flipped\_images() 对训练样本做翻转处理，其实现如下，可以看到其实并没有对图片本身做翻转，而是计算翻转后的标注框的坐标，然后用flipped:true标记是否做翻转处理。

代码段7：

|  |
| --- |
| # 对5011张训练集图片的标注框做左右翻转，需要处理的是标记框中的x坐标要做处理。处理后得到5011x2张训练集图片信息  def append\_flipped\_images(self):  num\_images = self.num\_images #5011  widths = self.\_get\_widths() #widths：list, len5011, voc2007中每张训练图片的宽度  for i in xrange(num\_images):#  boxes = self.roidb[i]['boxes'].copy() #调用pascal\_voc 中的gt\_roidb读取voc2007 pkl文件，self.roidb返回的是一个list，大小为5011， 这里获取第i图片的标记框坐标  oldx1 = boxes[:, 0].copy() #获取框的左上角 x 坐标，这里有3个boxes, 因此oldx1长度为3，[262, 164, 240]  oldx2 = boxes[:, 2].copy()#获取框的右下角 x 坐标，这里有3个boxes, 因此oldx1长度为3，[323, 252, 294]  boxes[:, 0] = widths[i] - oldx2 - 1 #宽度减去右下角 x 坐标，作为框的左上角x 坐标，即将图片左右翻转  boxes[:, 2] = widths[i] - oldx1 - 1 #宽度减去左上角 x 坐标，作为框的右下角 x 坐标，即将图片左右翻转  assert (boxes[:, 2] >= boxes[:, 0]).all() #右下角 x 坐标 要比 左上角 x 坐标 值要大  entry = {'boxes' : boxes,  'gt\_overlaps' : self.roidb[i]['gt\_overlaps'],  'gt\_classes' : self.roidb[i]['gt\_classes'],  'flipped' : True}  self.roidb.append(entry) #将翻转后的图片加入到roidb队列  self.\_image\_index = self.\_image\_index \* 2 #图片名称复制一份 |

再来看代码段6中的第10行rdl\_roidb.prepare\_roidb(imdb) 这个函数获取扩充后的数据集的图片路径、图片宽高、框的类别等，其在lib\roi\_data\_layer\roidb.py中实现如下：

代码段8：

|  |
| --- |
| def prepare\_roidb(imdb):  """Enrich the imdb's roidb by adding some derived quantities that  are useful for training. This function precomputes the maximum  overlap, taken over ground-truth boxes, between each ROI and  each ground-truth box. The class with maximum overlap is also  recorded.  """  sizes = [PIL.Image.open(imdb.image\_path\_at(i)).size  for i in xrange(imdb.num\_images)]#10022（原来5011张，翻转后10022张）张训练图片的宽高  roidb = imdb.roidb #10022张训练图片的标注框信息  for i in xrange(len(imdb.image\_index)):  roidb[i]['image'] = imdb.image\_path\_at(i) #图片的路径，第一张图为000005.jpg，该图中标注有3个框，都是chair(类别序号为9)  roidb[i]['width'] = sizes[i][0] #图片的宽  roidb[i]['height'] = sizes[i][1] #图片的高  # need gt\_overlaps as a dense array for argmax  gt\_overlaps = roidb[i]['gt\_overlaps'].toarray() #3x21, 有3个框，一行表示1个框，每列表示该框的类别，值为1，表示框的类别  # max overlap with gt over classes (columns)  max\_overlaps = gt\_overlaps.max(axis=1) #获取3x21中行方向（水平方向）中的最大值这里得到[1,1,1]  # gt class that had the max overlap  max\_classes = gt\_overlaps.argmax(axis=1) #获取行方向（水平）中最大值的列序号，这里得到[9,9,9]# 框的类别  roidb[i]['max\_classes'] = max\_classes # 框的类别  roidb[i]['max\_overlaps'] = max\_overlaps  # sanity checks  # max overlap of 0 => class should be zero (background)  zero\_inds = np.where(max\_overlaps == 0)[0] #zero\_inds空  assert all(max\_classes[zero\_inds] == 0)  # max overlap > 0 => class should not be zero (must be a fg class)  nonzero\_inds = np.where(max\_overlaps > 0)[0] #[0,1,2]  assert all(max\_classes[nonzero\_inds] != 0) |

在prepare\_roidb中遍历扩充后的10022张训练集图片，获取图片的路径、宽高，标注框的类别，保存到imdb的roidb中。至此代码段2分析完毕。

1. Caffe 添加自定义层：C++层

这里我们以mnist example为例来说明，如何在网络中添加自定义的Caffe层（C++ 实现）：

（1）不带参数的层

1.1网络文件添加自定义层

在caffe-master\examples\mnist\下拷贝lenet\_train\_test.prototxt，重命名为lenet\_train\_test\_new.prototxt

在第1个卷积层，即conv1层前添加一个自定义层，层名为NewLayer，层类型为New，如下：

|  |
| --- |
| layer {  name:"NewLayer"  type:"New"  bottom: "data"  top:"data"  } |

这里我们定义了一个最精简的层，该层没有配置参数，data从bottom进来，又从top出去，没做任何其他操作。由于这个层没有配置参数，因此仅在Caffe中添加hpp文件及cpp文件即可，不需要修改caffe.proto文件。

1.2添加hpp文件

在Caffe的VS工程里， libCaffe——inclulde——layers下添加一个new\_layer.hpp文件，文件的保存路径为：caffe-master\include\caffe\layers，代码如下：

|  |
| --- |
| #ifndef CAFFE\_NEW\_LAYER\_HPP\_  #define CAFFE\_NEW\_LAYER\_HPP\_    #include <vector>    #include "caffe/blob.hpp"  #include "caffe/layer.hpp"  #include "caffe/proto/caffe.pb.h"    namespace caffe {  template <typename Dtype>  class NewLayer : public Layer<Dtype> {  public:  explicit NewLayer(const LayerParameter& param)  : Layer<Dtype>(param) {}  virtual void LayerSetUp(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top){};  virtual void Reshape(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top){};    protected:  virtual void Forward\_cpu(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top);  virtual void Backward\_cpu(const vector<Blob<Dtype>\*>& top,  const vector<bool>& propagate\_down, const vector<Blob<Dtype>\*>& bottom);  virtual void Forward\_gpu(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top){};  virtual void Backward\_gpu(const vector<Blob<Dtype>\*>& top,  const vector<bool>& propagate\_down, const vector<Blob<Dtype>\*>& bottom){};  };    }    #endif |

****1.3添加cpp文件****

在Caffe的VS工程里， libCaffe——src——layers下添加一个new\_layer.cpp文件，文件的保存路径为：caffe-master\src\caffe\layers，代码如下：

|  |
| --- |
| #include <vector>  #include "caffe/layers/new\_layer.hpp"    namespace caffe {    template <typename Dtype>  void NewLayer<Dtype>::Forward\_cpu(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top) {  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  Dtype\* top\_data = top[0]->mutable\_cpu\_data();  const int count = bottom[0]->count();  for (int i = 0; i < count; ++i) {  top\_data[i] = bottom\_data[i];  }  }    template <typename Dtype>  void NewLayer<Dtype>::Backward\_cpu(const vector<Blob<Dtype>\*>& top,  const vector<bool>& propagate\_down,  const vector<Blob<Dtype>\*>& bottom) {  if (propagate\_down[0]) {  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  const Dtype\* top\_diff = top[0]->cpu\_diff();  Dtype\* bottom\_diff = bottom[0]->mutable\_cpu\_diff();  const int count = bottom[0]->count();  for (int i = 0; i < count; ++i) {  bottom\_diff[i] = top\_diff[i];  }  }  }    #ifdef CPU\_ONLY  STUB\_GPU(NewLayer);  #endif    INSTANTIATE\_CLASS(NewLayer); //类名，注：这个类名与prototxt文件中的层名不需一致  REGISTER\_LAYER\_CLASS(New); // 对应层的类型    } // namespace caffe |

以上即完成新层的添加，然后编译caffe项目（不用编译整个solution, 仅编译project, 编译caffe时会自动先编译libcaffe），在\caffe-master\Build\x64\Release下即生成新的caffe.lib及caffe.exe文件，然后对以上新的lenet\_train\_test\_new.prototxt进行训练或测试（lenet\_solver.prototxt文件中net指定的路径也需要修改），均可执行成功。

（2）添加带参数的层

**2.1添加带参数自定义层**

这里我们添加带参数的层，即将上面的NewLayer层改写如下，增加了参数集合new\_param，其中包含参数coeff1与coeff2：

|  |
| --- |
| layer {  name:"NewLayer"  type:"New"  bottom: "data"  top:"data"  new\_param {  coeff1: 1.0  coeff2: 2.0  }  } |

new\_layer.cpp增加获取参数的语句，如下：

|  |
| --- |
| #include <vector>  #include "caffe/layers/new\_layer.hpp"    namespace caffe {    template <typename Dtype>  void NewLayer<Dtype>::Forward\_cpu(const vector<Blob<Dtype>\*>& bottom,  const vector<Blob<Dtype>\*>& top) {  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  Dtype\* top\_data = top[0]->mutable\_cpu\_data();  const int count = bottom[0]->count();    float coeff = this->layer\_param\_.new\_param().coeff1(); // 获取参数  LOG(INFO) << "NewLayer, Forward\_cpu:" << coeff;    for (int i = 0; i < count; ++i) {  top\_data[i] = bottom\_data[i];  }  }    template <typename Dtype>  void NewLayer<Dtype>::Backward\_cpu(const vector<Blob<Dtype>\*>& top,  const vector<bool>& propagate\_down,  const vector<Blob<Dtype>\*>& bottom) {    float coeff = this->layer\_param\_.new\_param().coeff2(); // 获取参数  LOG(INFO) << "NewLayer, Backward\_cpu:" << coeff;    if (propagate\_down[0]) {  const Dtype\* bottom\_data = bottom[0]->cpu\_data();  const Dtype\* top\_diff = top[0]->cpu\_diff();  Dtype\* bottom\_diff = bottom[0]->mutable\_cpu\_diff();  const int count = bottom[0]->count();  for (int i = 0; i < count; ++i) {  bottom\_diff[i] = top\_diff[i];  }  }  }    #ifdef CPU\_ONLY  STUB\_GPU(NewLayer);  #endif    INSTANTIATE\_CLASS(NewLayer); //类名，对应prototxt文件中的层名  REGISTER\_LAYER\_CLASS(New); // 对应层的类型    } // namespace caffe |

****2.2 修改caffe.proto****

修改caffe-master\src\caffe\proto\caffe.proto，有两处修改点：

在message LayerParameter下添加：

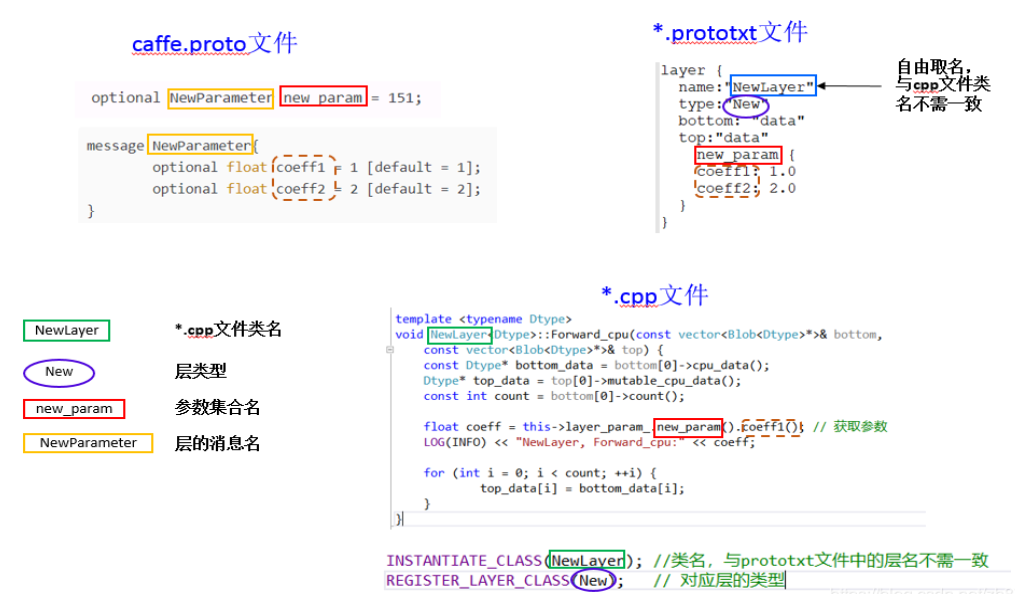
|  |
| --- |
| optional NewParameter new\_param = 151; |

注意设定的ID值不能与其他已设置的有相同，而message LayerParameter上面注明了可以设定的、没有冲突的ID值。

再在其他位置添加一个NewLayer层的message函数：

|  |
| --- |
| message NewParameter{  optional float coeff1 = 1 [default = 1];  optional float coeff2 = 2 [default = 2];  } |

注意各个文件中参数的命名要匹配一致，如下：

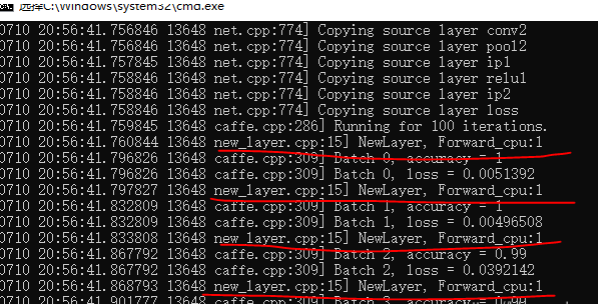


修改好后，在VS中编译caffe.exe， 编译过程中caffe工程会根据caffe.proto生成新的caffe.pb.c与caffe.pb.h文件，分别位于caffe-master\src\caffe\proto\目录与caffe-master\include\caffe\proto\目录下，如果发现这两个文件没有更新，则可能是caffe.ptoto文件编写出错，例如，我一开始将optional float coeff1 = 1 [default = 1]中的 "default" 写成了 "defalut"，导致一直不生成新的caffe.pb.c与caffe.pb.h文件，caffe也无法编译成功，检查很久后才发现问题所在。

在caffe-master目录下运行test\_mnist.bat文件，文件内容：

|  |
| --- |
| .\Build\x64\Release\caffe.exe test -model=examples\mnist\lenet\_train\_test\_new.prototxt -weights=examples\mnist\mnist\_data\lenet\_iter\_10000.caffemodel -iterations=100  pause |

输出如下，可以看到打印出了NewLayer层的LOG输出，即表示添加新层成功。



 编译caffe.proto无法生成新的caffe.pb.c与caffe.pb.h的问题

如果无法生成caffe.pb.c与caffe.pb.h，那就是caffe.proto文件有错误，可以用protobuf工具检查文件哪里出错：

到以下路径下载protubuf，我这里下载 protoc-3.9.0-rc-1-win32.zip

https://github.com/protocolbuffers/protobuf/releases

下载解压，将bin\protoc.exe添加到系统环境变量PATH中

在caffe-master\src\caffe\proto\目录下创建bat文件，如下：

|  |
| --- |
| protoc caffe.proto --cpp\_out=./  protoc caffe.proto --python\_out=./  md ..\..\..\python\caffe\proto\  copy /y .\caffe\_pb2.py ..\..\..\python\caffe\proto\  copy nul ..\..\..\python\caffe\proto\\_\_init\_\_.py  pause |

双击运行，如果caffe.proto文件没有错误，会在caffe.proto的目录下生成caffe.pb.c及caffe.proto.h文件，如果错误，会显示出错的地方。

1. Caffe添加自定义层：Python层

前面一篇讲述了如何添加自定义的Caffe C++层，本篇讲解如何添加自定义的Python层，依然以mnist example为例子，在caffe-master\examples\mnist中的 lenet\_train\_test.ptototxt文件中，conv1层前添加以下测试层，这个层对网络不做任何修改，仅用于测试。

|  |
| --- |
| layer {  name: 'test'  type: 'Python'  bottom: 'data'  top:'data'  python\_param {  module: 'test\_layer' #与python文件名相同  layer: 'MyLayer' #与python文件中的类名相同  param\_str: "'num\_classes': 10"  }  } |

这个层的类型为python，我们需要编写python文件来实现这个层的内容。在caffe-master\examples\mnist目录下创建一个文件夹python\_layers文件夹，在python\_layers文件夹中创建test\_layer.py, 类名为MyLayer，如下，包含setup，reshape, forward, backward 4个函数，每个函数的输入输出都是bottom及top，对应prototxt文件的bottom及top，bottom与top均是Blob数据类型。

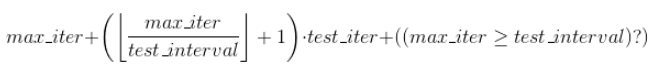
|  |
| --- |
| import caffe    class MyLayer(caffe.Layer):  #初始化时调用，检查输入的参数是否异常  def setup(self, bottom, top):  print 'setup'    #初始化时、前向传播时调用，用于设定参数的siaze  def reshape(self, bottom, top):  print 'reshape'    #前向传播时调用  def forward(self, bottom, top):  print 'forward'    #反向传播，如果本层不需要反向传播，则不调用  def backward(self, bootom, top):  print 'backward' |

以上是最简单的写法，各个函数仅打印了函数的名。我们先不详细实现各个函数的功能，我们先运行这个Python层，先看能否运行成功。

在Caffe根目录下，即caffe-master\下创建train\_mnist.py文件，注意这里要在caff-master目录下创建，因为mnist的\*.prototxt文件中设置的路径是在examples开始。

|  |
| --- |
| import os.path as osp  import sys  import caffe  this\_dir = osp.dirname(\_\_file\_\_)  layerpath = osp.join(this\_dir, 'examples/mnist/python\_layers')  sys.path.insert(0, layerpath) #将python文件的位置添加到环境变量    caffe.set\_mode\_gpu()  caffe.set\_device(0)    solver = caffe.SGDSolver('./examples/mnist/lenet\_solver.prototxt')  solver.solve() |

运行以上python文件，通过打印的log可以看到在初始化时，'setup'和'reshape' 各打印了2次，然后在训练过程中，有多少次向前计算则‘reshape’与‘forward’就被调用多少次，而‘forward没有被调用’。通过实验测试，前向计算的次数为：



\left (\left \lfloor\frac{max\_iter}{test\_interval} \right \rfloor+1 \right )\cdot test\_iter表示验证测试前向计算的次数，因为第0次时要做一次前向验证测试，因此要加1，后面的表示当max\_iter大于等于test\_interval时式子为1，否则为0。例如当max\_iter为10000，test\_interval为500，test\_iter为100时，前向计算的次数为12101次。

下面逐步来实现各层。

**setup 函数**

|  |
| --- |
| def setup(self, bottom, top):  layer\_params = yaml.load(self.param\_str)  self.\_num\_classes = layer\_params['num\_classes']  print 'setup, bottom len:', len(bottom), ', top len:', len(top), ', class num:', self.\_num\_classes |

注意这里要在py文件头部import yaml，这里获取了.prototxt文件设置的参数param\_str。另外这里len(top)与len(bottom)表示输入输出的参数个数。如果层文件设置为：

|  |
| --- |
| layer {  name: 'test'  type: 'Python'  bottom: 'data'  bottom: 'label'  top:'data'  top:'label'  python\_param {  module: 'test\_layer'  layer: 'MyLayer'  param\_str: "'num\_classes': 10"  }  } |

这里bottom和top的参数都有多个，通过序号的方式分别获取与设置bottom与top的参数，如下：

|  |
| --- |
| top[0].reshape(\*bottom[0].data.shape) #data  top[1].reshape(\*bottom[1].data.shape) # label |

**reshape 函数:**

这里reshape直接写作如下：

|  |
| --- |
| def reshape(self, bottom, top):  top[0].reshape(\*bottom[0].data.shape) #data |

**forward函数:**

|  |
| --- |
| **def forward(self, bottom, top):**  **top[0].data[...] = bottom[0].data[:]** |

**backward函数:**

|  |
| --- |
| **def backward(self, bootom, top):**  **for i in range(len(propagate\_down)):**  **if not propagate\_down[i]:**  **continue**  **bottom[i].diff[...] = top[i].diff[:]** |