Caffe学习总结（更新中）

主要参考一个大牛的caffe学习系列博客：<http://www.cnblogs.com/denny402/tag/caffe/>

1. *关于caffe的简介*

caffe是一个清晰而高效的深度学习框架，纯粹的C++/CUDA架构，支持命令行、Python和MATLAB接口，可以在CPU和GPU直接无缝切换；

caffe的主要优势：

（1）CPU与GPU的无缝切换；

（2）模型与优化都是通过配置文件来设置，无需代码；

简洁好用不多说。

1. *Caffe环境的搭建，参考另一篇文章：《安装caffe》。*
2. *Caffe总体框架介绍*

主要参考：[深度学习框架Caffe源码解析](http://mp.weixin.qq.com/s/rWDCYO5k06zT9BcXk21JJg)；

若想详细了解caffe源码，参考官方教程[Tutorial Documentation](http://caffe.berkeleyvision.org/tutorial) 或者看《caffe官方教程中译本》。

还有一个博客写的很好，caffe源码解析：<http://www.cnblogs.com/louyihang-loves-baiyan/>

Caffe主要由Blob，Layer，Net 和 Solver这几个部分组成。

1）Blob

主要用来表示网络中的数据，包括训练数据，网络各层自身的参数(包括权值、偏置以及它们的梯度)，网络之间传递的数据都是通过 Blob 来实现的，同时 Blob 数据也支持在 CPU 与 GPU 上存储，能够在两者之间做同步。

2）Layer

是对神经网络中各种层的一个抽象，包括我们熟知的卷积层和下采样层，还有全连接层和各种激活函数层等等。同时每种 Layer 都实现了前向传播和反向传播，并通过 Blob 来传递数据。

3） Net

是对整个网络的表示，由各种 Layer 前后连接组合而成，也是我们所构建的网络模型。

4） Solver

定义了针对 Net 网络模型的求解方法，记录网络的训练过程，保存网络模型参数，中断并恢复网络的训练过程。自定义 Solver 能够实现不同的网络求解方式。

完整巧妙不多说。

1. *总流程（重点）*

完成一个简单的自己的网络模型训练预测，主要包含几个步骤：（以一个实例介绍）

1. 准备数据；

例1，学习系列博客上的数据，后面会用到，这里简单说明一下：

可以从这里下载：<http://pan.baidu.com/s/1nuqlTnN>

共有500张图片，分为大巴车、恐龙、大象、鲜花和马五个类，每个类100张。编号分别以3，4，5，6，7开头，各为一类。从其中每类选出20张作为测试，其余80张作为训练。因此最终训练图片400张，测试图片100张，共5类。我将图片放在caffe根目录下的data文件夹下面。即训练图片目录：data/re/train/ ,测试图片目录: data/re/test/。

1. 数据格式处理，也就是把我们jpg,jpeg,png,tif等格式的图片（可能存在大小不一致的问题），处理转换成caffe中能够运行的db（leveldb/lmdb)文件。

参考学习系列博客

[Caffe学习系列(11)：图像数据转换成db（leveldb/lmdb)文件](http://www.cnblogs.com/denny402/p/5082341.html)

讲的非常详细。

总结一下，在caffe中，作者为我们提供了这样一个文件：**convert\_imageset.cpp**，存放在根目录下的tools文件夹下。编译之后，生成对应的可执行文件放在 buile/tools/ 下面，这个文件的作用就是用于将图片文件转换成caffe框架中能直接使用的db文件。

该文件的使用格式：

**convert\_imageset [FLAGS] ROOTFOLDER/ LISTFILE DB\_NAME**

四个参数：

**FLAGS:** 图片参数组，后面详细介绍；

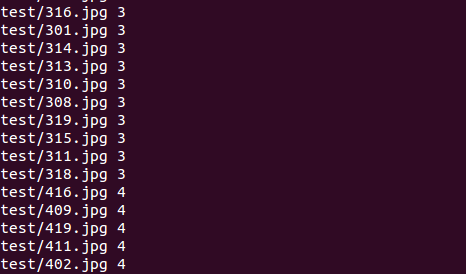
**ROOTFOLDER/:** 图片存放的绝对路径，从linux系统根目录开始；

**LISTFILE:** 图片文件列表清单，一般为一个txt文件，一行一张图片；

**DB\_NAME:** 最终生成的db文件存放目录。

其中第二个和第四个目录是自己决定的，不多说；

先说第三个，所谓图片列表清单，也叫标签文件，一般该文件存放图片文件路径，以及该图片的标签（属于哪个类）;一般来说，标签文件有两个，一个描述训练集合-train.txt，一个描述测试集合-test.txt，（可能还有描述验证的val.txt）,例1中测试图片转换后的标签文件格式（test.txt）如下：



那怎么由图片生成标签文件呢？

当然图片很少的时候，直接手动输入就好了，但图片很多的情况，就需要用脚本文件来自动生成了。之前对脚本的编写不是很熟，有个快速上手的文章：

<http://www.cnblogs.com/handsomecui/p/5869361.html>（其实非常简单(￣▽￣)）

对于例1，在examples下面创建一个myfile的文件夹，来用存放配置文件和脚本文件。然后编写一个脚本create\_filelist.sh，用来生成train.txt和test.txt清单文件。脚本（/examples/myfile/create\_filelist.sh）编写如下：

#!/usr/bin/env sh

DATA=data/re/

MY=examples/myfile

echo "Create train.txt..."

rm -rf $MY/train.txt

for i in 3 4 5 6 7

do

find $DATA/train -name $i\*.jpg | cut -d '/' -f4-5 | sed "s/$/ $i/">>$MY/train.txt

done

echo "Create test.txt..."

rm -rf $MY/test.txt

for i in 3 4 5 6 7

do

find $DATA/test -name $i\*.jpg | cut -d '/' -f4-5 | sed "s/$/ $i/">>$MY/test.txt

done

echo "All done"

生成的.txt文件，就可以作为第三个参数，直接使用了。

最后了解一下FLAGS这个参数组，有些什么内容：

**-gray:** 是否以灰度图的方式打开图片。程序调用opencv库中的imread()函数来打开图片，默认为false

**-shuffle:** 是否随机打乱图片顺序。默认为false

**-backend:**需要转换成的db文件格式，可选为leveldb或lmdb,默认为lmdb

**-resize\_width/resize\_height:** 改变图片的大小。在运行中，要求所有图片的尺寸一致，因此需要改变图片大小。 程序调用opencv库的resize（）函数来对图片放大缩小，默认为0，不改变

**-check\_size:** 检查所有的数据是否有相同的尺寸。默认为false,不检查

**-encoded:** 是否将原图片编码放入最终的数据中，默认为false

**-encode\_type:** 与前一个参数对应，将图片编码为哪一个格式：‘png','jpg'......

继续看例1，于是将训练数据转换成lmdb文件命令如下：



注：由于图片大小不一，因此这里统一转换成256\*256大小。

类似的，将测试数据转换成lmdb文件命令如下：



当然，由于参数比较多，依然可以写脚本（examples/myfile/create\_lmdb.sh），

#!/usr/bin/env sh

MY=examples/myfile

echo "Create train lmdb.."

rm -rf $MY/img\_train\_lmdb

build/tools/convert\_imageset \

--shuffle \

--resize\_height=256 \

--resize\_width=256 \

/home/xxx/caffe/data/re/ \

$MY/train.txt \

$MY/img\_train\_lmdb

echo "Create test lmdb.."

rm -rf $MY/img\_test\_lmdb

build/tools/convert\_imageset \

--shuffle \

--resize\_width=256 \

--resize\_height=256 \

/home/xxx/caffe/data/re/ \

$MY/test.txt \

$MY/img\_test\_lmdb

echo "All Done.."

这里的xxx是你具体的路径。

运行成功后，会在 examples/myfile下面生成两个文件夹img\_train\_lmdb和img\_test\_lmdb，分别用于保存图片转换后的lmdb文件。

1. 计算均值并保存

图片减去均值再训练，会提高训练速度和精度。因此，一般都会有这个操作。

caffe程序提供了一个计算均值的文件**compute\_image\_mean.cpp**，我们直接使用就可以了。

对于例1，命令如下：

sudo build/tools/compute\_image\_mean examples/myfile/img\_train\_lmdb examples/myfile/mean.binaryproto

compute\_image\_mean带两个参数，第一个参数是lmdb训练数据位置，第二个参数设定均值文件的名字及保存路径。

运行成功后，会在 examples/myfile/ 下面生成一个mean.binaryproto的均值文件。

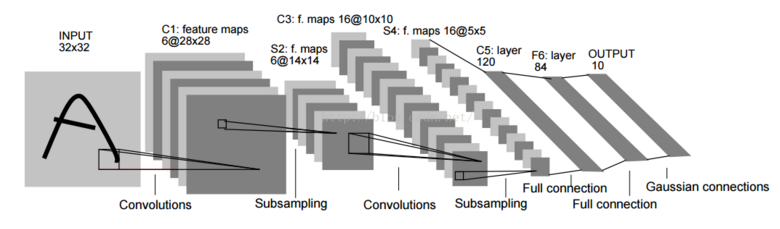
1. **创建模型**

在caffe中是通过.prototxt配置文件来定义网络模型。

例如，可以打开caffe自带的手写数字库MNIST例子的网络结构文件：

sudo vim examples/mnist/lenet\_train\_test.prototxt

这个网络模型是非常有名的**LeNet模型**，网络结构如下图。



可以看到各个网络层是如何定义的：

* 数据层（也叫输入层）

layer {

name: "mnist" //表示层名

type: "Data" //表示层的类型

top: "data"

top: "label"

include {

phase: TRAIN //表示仅在训练阶段起作用

}

transform\_param {

scale: 0.00390625 //将图像像素值归一化

}

data\_param {

source: "examples/mnist/mnist\_train\_lmdb" //数据来源

batch\_size: 64 //训练时每个迭代的输入样本数量

backend: LMDB //数据类型

}

}

关于数据层及参数的编写，参看学习系列（2）：[Caffe学习系列(2)：数据层及参数](http://www.cnblogs.com/denny402/p/5070928.html)

简单介绍一下其中参数：

name: 表示该层的名称，可随意取

type: 层类型，如果是Data，表示数据来源于LevelDB或LMDB。根据数据的来源不同，数据层的类型也不同（后面会详细阐述）。一般在练习的时候，我们都是采 用的LevelDB或LMDB数据，因此层类型设置为Data。

top或bottom: 每一层用bottom来输入数据，用top来输出数据。如果只有top没有bottom，则此层只有输出，没有输入。反之亦然。如果有多个 top或多个bottom，表示有多个blobs数据的输入和输出。

data 与 label: 在数据层中，至少有一个命名为data的top。如果有第二个top，一般命名为label。 这种(data,label)配对是分类模型所必需的。

include: 一般训练的时候和测试的时候，模型的层是不一样的。该层（layer）是属于训练阶段的层，还是属于测试阶段的层，需要用include来指定。如果没有include参数，则表示该层既在训练模型中，又在测试模型中。

Transformations: 数据的预处理，可以将数据变换到定义的范围内。如设置scale为0.00390625，实际上就是1/255, 即将输入数据由0-255归一化到0-1之间。

* 视觉层

视觉层包括Convolution, Pooling, Local Response Normalization (LRN), im2col等层。

关于视觉层及参数的编写，参看：[Caffe学习系列(3)：视觉层（Vision Layers)及参数](http://www.cnblogs.com/denny402/p/5071126.html)

仍然以mnist为例，其**卷积层**定义如下：

layer {

name: "conv1"

type: "Convolution"

bottom: "data" //输入是data

top: "conv1" //输出是卷积特征

param {

lr\_mult: 1 //权重参数w的学习率倍数

}

param {

lr\_mult: 2 //偏置参数b的学习率倍数

}

convolution\_param {

num\_output: 20

kernel\_size: 5

stride: 1

weight\_filler { //权重参数w的初始化方案，使用xavier算法

type: "xavier"

}

bias\_filler {

type: "constant" //偏置参数b初始化化为常数，一般为0

}

}

}

简单介绍一下其中参数：

层类型：Convolution

lr\_mult: 学习率的系数，最终的学习率是这个数乘以solver.prototxt配置文件中的base\_lr。如果有两个lr\_mult, 则第一个表示权值的学习率，第二个表示偏置项的学习率。一般偏置项的学习率是权值学习率的两倍。

在后面的convolution\_param中，我们可以设定卷积层的特有参数。

必须设置的参数：

num\_output: 卷积核（filter)的个数

kernel\_size: 卷积核的大小。如果卷积核的长和宽不等，需要用kernel\_h和kernel\_w分别设定

其它参数：

stride: 卷积核的步长，默认为1。也可以用stride\_h和stride\_w来设置。

pad: 扩充边缘，默认为0，不扩充。 扩充的时候是左右、上下对称的，比如卷积核的大小为5\*5，那么pad设置为2，则四个边缘都扩充2个像素，即宽度和高度都扩充了4个像素,这样卷积运算之后的特征图就不会变小。也可以通过pad\_h和pad\_w来分别设定。

weight\_filler: 权值初始化。 默认为“constant",值全为0，很多时候我们用"xavier"算法来进行初始化，也可以设置为”gaussian"

bias\_filler: 偏置项的初始化。一般设置为"constant",值全为0。

bias\_term: 是否开启偏置项，默认为true, 开启

group: 分组，默认为1组。如果大于1，我们限制卷积的连接操作在一个子集内。如果我们根据图像的通道来分组，那么第i个输出分组只能与第i个输入分组进行连接。

输入：n\*c0\*w0\*h0

输出：n\*c1\*w1\*h1

其中，c1就是参数中的num\_output，生成的特征图个数

w1=(w0+2\*pad-kernel\_size)/stride+1;

h1=(h0+2\*pad-kernel\_size)/stride+1;

如果设置stride为1，前后两次卷积部分存在重叠。如果设置pad=(kernel\_size-1)/2,则运算后，宽度和高度不变。

**Pooling层，**也叫池化层，为了减少运算量和数据维度而设置的一种层。

LeNet总共有两个，其中一个定义如下：

layer {

name: "pool1"

type: "Pooling"

bottom: "conv1"

top: "pool1"

pooling\_param {

pool: MAX

kernel\_size: 2

stride: 2

}

}

层类型：Pooling

必须设置的参数：

　kernel\_size: 池化的核大小。也可以用kernel\_h和kernel\_w分别设定。

其它参数：

　pool: 池化方法，默认为MAX。目前可用的方法有MAX, AVE, 或STOCHASTIC

　　pad: 和卷积层的pad的一样，进行边缘扩充。默认为0

　　stride: 池化的步长，默认为1。一般我们设置为2，即不重叠。也可以用stride\_h和stride\_w来设置。

还有Local Response Normalization (LRN)层和im2col层等，在LeNet中没有定义，可以参看AlexNet或GoogLenet等更复杂的模型。

* 激活层

关于视觉层及参数的编写，参看[Caffe学习系列(4)：激活层（Activiation Layers)及参数](http://www.cnblogs.com/denny402/p/5072507.html)

在LeNet中，用到的激活函数是ReLU / Rectified-Linear and Leaky-ReLU，

ReLU是目前使用最多的激活函数，主要因为其收敛更快，并且能保持同样效果。标准的ReLU函数为max(x, 0)，当x>0时，输出x; 当x<=0时，输出0。

f(x)=max(x,0)。

定义如下：

layer {

name: "relu1"

type: "ReLU"

bottom: "ip1"

top: "ip1"

}

层类型：ReLU

可选参数：

negative\_slope：默认为0. 对标准的ReLU函数进行变化，如果设置了这个值，那么数据为负数时，就不再设置为0，而是用原始数据乘以negative\_slope。

RELU层支持in-place计算，这意味着bottom的输出和输入相同以避免内存的消耗。

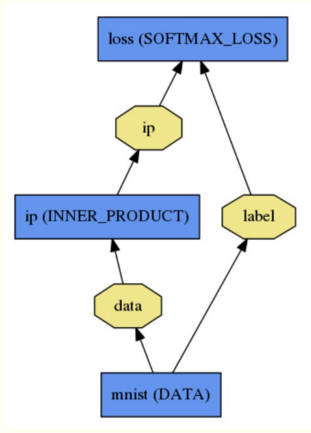
* 其他层

包括：softmax\_loss层，Inner Product层，accuracy层，reshape层和dropout层等，参看[**Caffe学习系列(5)：其它常用层及参数**](http://www.cnblogs.com/denny402/p/5072746.html)，不多说。

* 关于模型编写的一个总结：

[**Caffe学习系列(6)：Blob,Layer and Net以及对应配置文件的编写**](http://www.cnblogs.com/denny402/p/5073427.html)

一个完整网络例子：



第一层：name为mnist, type为Data，没有输入（bottom)，只有两个输出（top),一个为data,一个为label

第二层：name为ip，type为InnerProduct, 输入数据data, 输出数据ip

第三层：name为loss, type为SoftmaxWithLoss，有两个输入，一个为ip,一个为label，有一个输出loss,没有画出来。

对应的配置文件prototxt就可以这样写：

name: "LogReg"

layer {

name: "mnist"

type: "Data"

top: "data"

top: "label"

data\_param {

source: "input\_leveldb"

batch\_size: 64

}

}

layer {

name: "ip"

type: "InnerProduct"

bottom: "data"

top: "ip"

inner\_product\_param {

num\_output: 2

}

}

layer {

name: "loss"

type: "SoftmaxWithLoss"

bottom: "ip"

bottom: "label"

top: "loss"

}

1. **编写配置文件**

主要参看[**Caffe学习系列(7)：solver及其配置**](http://www.cnblogs.com/denny402/p/5074049.html) 和[**Caffe学习系列(8)：solver优化方法**](http://www.cnblogs.com/denny402/p/5074212.html)

先看一下mnist中配置文件（一般是..solver.prototxt文件）：

sudo vim examples/mnist/lenet\_solver.prototxt

配置文件如下：

# The train/test net protocol buffer definition

net: "examples/mnist/lenet\_train\_test.prototxt"  *//设置深度网络模型，就是介绍的模型。*

# test\_iter specifies how many forward passes the test should carry out.

# In the case of MNIST, we have test batch size 100 and 100 test iterations,

# covering the full 10,000 testing images.

test\_iter: 100  *//这个要与test layer中的batch\_size结合起来理解。mnist数据中测试样本总数为10000，一次性执行全部数据效率很低，因此我们将测试数据分成几个批次来执行，每个批次的数量就是batch\_size。假设我们设置batch\_size为100，则需要迭代100次才能将10000个数据全部执行完。因此test\_iter设置为100。执行完一次全部数据，称之为一个epoch*

# Carry out testing every 500 training iterations.

test\_interval: 500  *// 测试间隔。也就是每训练500次，才进行一次测试。*

# The base learning rate, momentum and the weight decay of the network.

base\_lr: 0.01

momentum: 0.9  *//上一次梯度更新的权重*

weight\_decay: 0.0005  *//权重衰减项，防止过拟合的一个参数。*

# The learning rate policy

lr\_policy: "inv"

gamma: 0.0001

power: 0.75 *//这几个参数用于学习率的设置。只要是梯度下降法来求解优化，都会有一个学习率，也叫步长。base\_lr用于设置基础学习率，在迭代的过程中，可以对基础学习率进行调整。怎么样进行调整，就是调整的策略，由lr\_policy来设置*。

# Display every 100 iterations

display: 100 *//每100次迭代显示一次*

# The maximum number of iterations

max\_iter: 10000  *//最大迭代次数。这个数设置太小，会导致没有收敛，精确度很低。设置太大，会导致震荡，浪费时间。*

# snapshot intermediate results

snapshot: 5000

snapshot\_prefix: "examples/mnist/lenet"

*//快照。将训练出来的model和solver状态进行保存，snapshot用于设置训练多少次后进行保存，默认为0，不保存。snapshot\_prefix设置保存路径。*

# solver mode: CPU or GPU

solver\_mode: GPU  *//设置运行模式。默认为GPU,如果你没有GPU,则需要改成CPU,否则会出错。*

其实英文注释将参数的意思说的很清楚，我自己也在后面注释了，具体详细的解释还是参看前面提到的两个链接。

优化的方法不是很清楚，这里不多说。

继续看例1，简单一点，直接用程序自带的caffenet模型，位置在 models/bvlc\_reference\_caffenet/文件夹下, 将需要的两个配置文件，复制到myfile文件夹内：

sudo cp models/bvlc\_reference\_caffenet/solver.prototxt examples/myfile/

sudo cp models/bvlc\_reference\_caffenet/train\_val.prototxt examples/myfile/

修改其中的solver.prototxt

sudo vi examples/myfile/solver.prototxt

修改如下：

net: "examples/myfile/train\_val.prototxt"

test\_iter: 2

test\_interval: 50

base\_lr: 0.001

lr\_policy: "step"

gamma: 0.1

stepsize: 100

display: 20

max\_iter: 500

momentum: 0.9

weight\_decay: 0.005

solver\_mode: GPU

说明：100个测试数据，batch\_size为50，因此test\_iter设置为2，就能全cover了。在训练过程中，调整学习率，逐步变小。

修改train\_val.protxt，只需要修改两个阶段的data层就可以了，其它可以不用管。

name: "CaffeNet"

layer {

name: "data"

type: "Data"

top: "data"

top: "label"

include {

phase: TRAIN

}

transform\_param {

mirror: true

crop\_size: 227

mean\_file: "examples/myfile/mean.binaryproto"

}

data\_param {

source: "examples/myfile/img\_train\_lmdb"

batch\_size: 256

backend: LMDB

}

}

layer {

name: "data"

type: "Data"

top: "data"

top: "label"

include {

phase: TEST

}

transform\_param {

mirror: false

crop\_size: 227

mean\_file: "examples/myfile/mean.binaryproto"

}

data\_param {

source: "examples/myfile/img\_test\_lmdb"

batch\_size: 50

backend: LMDB

}

}

实际上就是修改两个data layer的mean\_file和source这两个地方，其它都没有变化 。

1. 训练和测试

caffe的运行提供三种接口：c++接口（命令行）、python接口和matlab接口。

这里选择命令行，参看[**Caffe学习系列(10)：命令行解析**](http://www.cnblogs.com/denny402/p/5076285.html)**。**

回到例1，命令很简单：

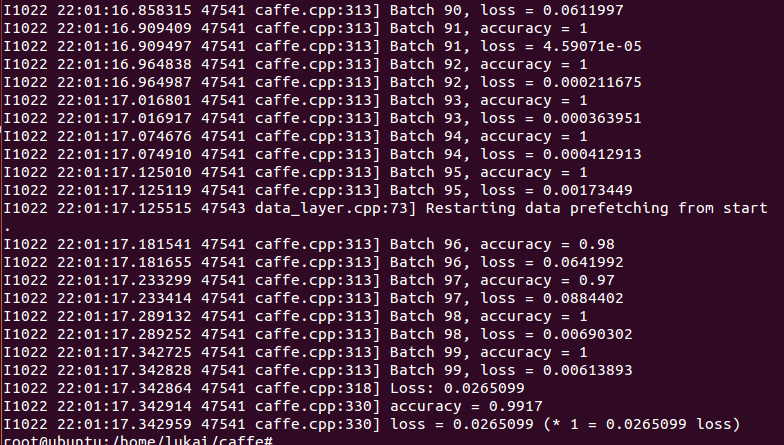
sudo build/tools/caffe train -solver examples/myfile/solver.prototxt

运行时间和最后的精确度，会根据机器配置，参数设置的不同而不同。gpu+cudnn的配置运行500次，大约8分钟，精度为95%。

其他的例子，参看[**Caffe学习系列(9)：运行caffe自带的两个简单例子**](http://www.cnblogs.com/denny402/p/5075490.html)

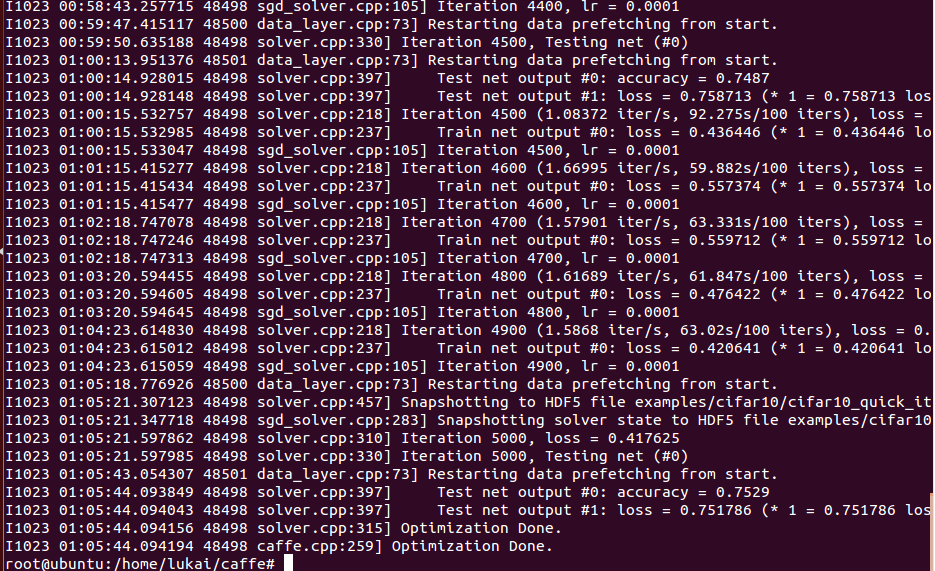
运行结果大致（我自己机子上跑的结果）如下：

mnist:



精度99%左右。

cifar10:



精度75%左右。

1. 用训练好的model用到自己数据上

参看[**Caffe学习系列(23)：如何将别人训练好的model用到自己的数据上**](http://www.cnblogs.com/denny402/p/5137534.html)

* 下载caffemodel

参看：[Model-Zoo](https://github.com/BVLC/caffe/wiki/Model-Zoo)或[model zoo documentation](http://caffe.berkeleyvision.org/model_zoo.html)；

下载地方：<https://github.com/BVLC/caffe/tree/master/models>

简单点，直接运行脚本来下载：

sudo ./scripts/download\_model\_binary.py <dirname>

<dirname>可以是：

models/bvlc\_reference\_caffenet

models/bvlc\_alexnet

models/bvlc\_reference\_rcnn\_ilsvrc13

models/bvlc\_googlenet

例2，这里我们以Alexnet为例，

sudo ./scripts/download\_model\_binary.py models/bvlc\_alexnet

下载后，可以在models/bvlc\_alexnet/下看到.caffemodel文件。



* 小插曲，先做一个简单的测试，单图的测试，也就是分类。

参考[**Caffe学习系列(20)：用训练好的caffemodel来进行分类**](http://www.cnblogs.com/denny402/p/5111018.html)

在caffe根目录下的 examples/cpp-classification/ 文件夹下面，有个**classification.cpp**文件，就是用来分类的。

使用该文件的格式如下：

./build/examples/cpp\_classification/classification.bin \

网络结构文件：xx/xx/deploy.prototxt \

训练的模型文件：xx/xx/xx.caffemodel \

训练的图像的均值文件：xx/xx/xx.binaryproto \

类别名称标签文件：xx/xx/synset\_words.txt \

待测试图像：xx/xx/xx.jpg

在例2中，即AlexNet模型中，

网络结构文件：models/bvlc\_alexnet/deploy.prototxt

训练的模型文件：models/bvlc\_alexnet/bvlc\_alexnet.caffemodel

训练的图像的均值文件：

直接用脚本下载：sudo sh ./data/ilsvrc12/get\_ilsvrc\_aux.sh

下载后的文件是data/ilsvrc12/imagenet\_mean.binaryproto

类别名称标签文件：

在调用脚本文件下载均值的时候，这个文件也一并下载好了。

文件：data/ilsvrc12/synset\_words.txt

待测试图像：

就以自带的测试图片吧：examples/images/cat.jpg

于是完整的命令是：

./build/examples/cpp\_classification/classification.bin \

models/bvlc\_alexnet/deploy.prototxt \

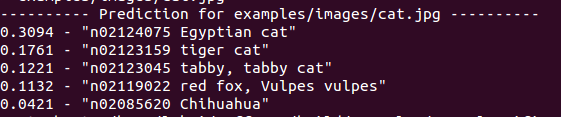
models/bvlc\_alexnet/bvlc\_alexnet.caffemodel \

data/ilsvrc12/imagenet\_mean.binaryproto \

data/ilsvrc12/synset\_words.txt \

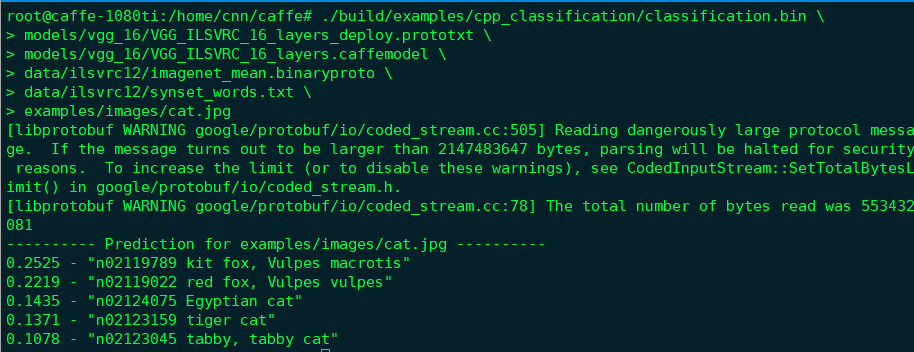
examples/images/cat.jpg

运行结果如下：



（不要在意细节= =）

换VGG16试试（在服务器上跑）：



（好吧，结果还是好奇怪。。）

* 继续，第二步依然是准备数据并处理啦，前面提过了，不在赘述。

还是用例1的数据。

* 接下来就是测试

依旧跟前面说的一样，要修改.prototxt文件：

先拷贝一份到myfile，顺便把.caffemodel也拷贝过来吧:

cp models/bvlc\_alexnet/bvlc\_alexnet.caffemodel examples/myfile/

cp models/bvlc\_alexnet/train\_val.prototxt examples/myfile/

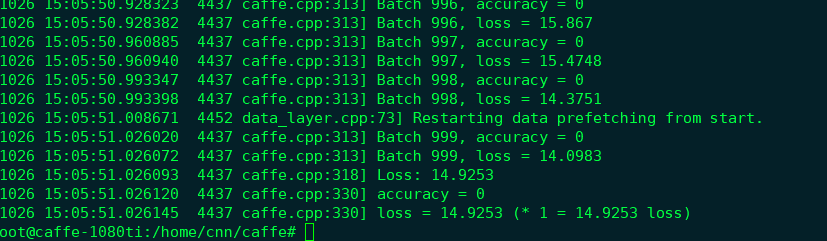
vim examples/myfile/train\_val.prototxt

还是修改两个data layer的mean\_file和source这两个地方，其它都没有变化 。

接着按照命令行（C++）的测试命令格式，（不清楚的话参考博客）输入测试命令就好了：

./build/tools/caffe test -model examples/myfile/train\_val.prototxt -weights examples/myfile/bvlc\_alexnet.caffemodel -gpu 0 -iterations 1000

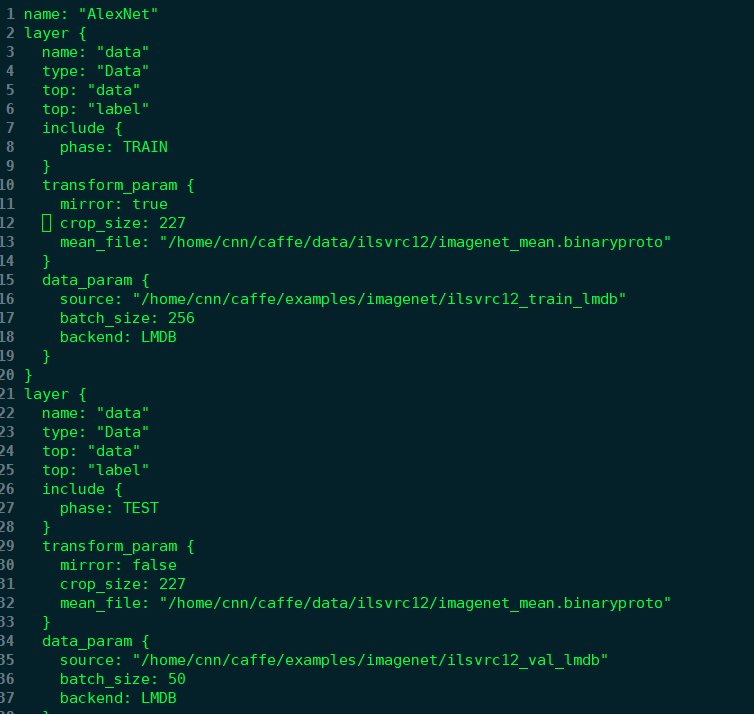
这个测试在服务器上完成的，结果如下：



额，这个结果很僵。。

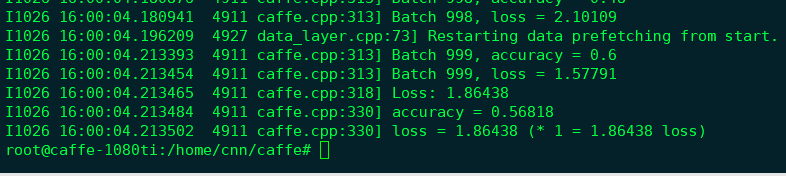
应该是数据集的问题，用服务器上处理好的数据集吧，

修改train\_val.prototxt:



再输入测试命令：

./build/tools/caffe test -model examples/myfile/train\_val.prototxt -weights examples/myfile/bvlc\_alexnet.caffemodel -gpu 0 -iterations 1000

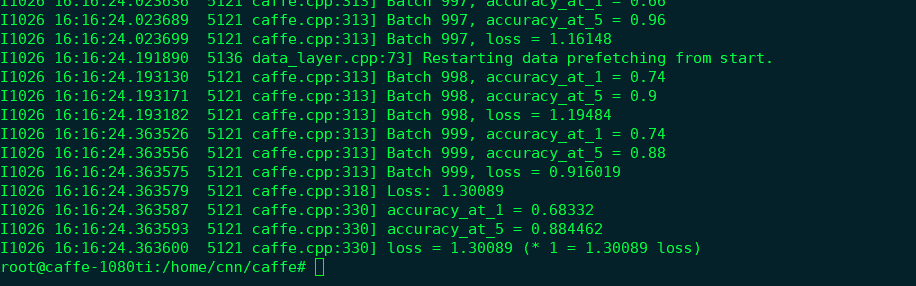


换个VGG试试：

按前面步骤（服务器上没有\_train\_val.prototxt文件，我自己找了一个，复制到examples/myfile/，按前面的修改），测试：

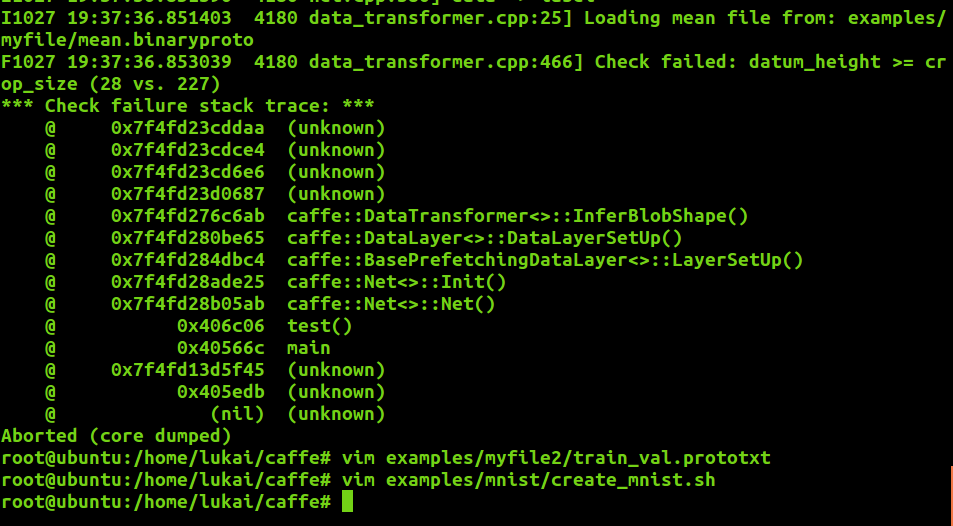


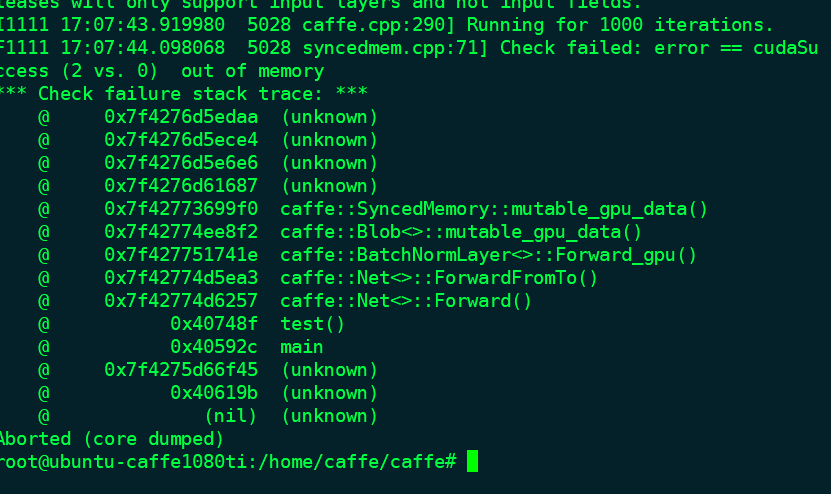
运行结果如下：



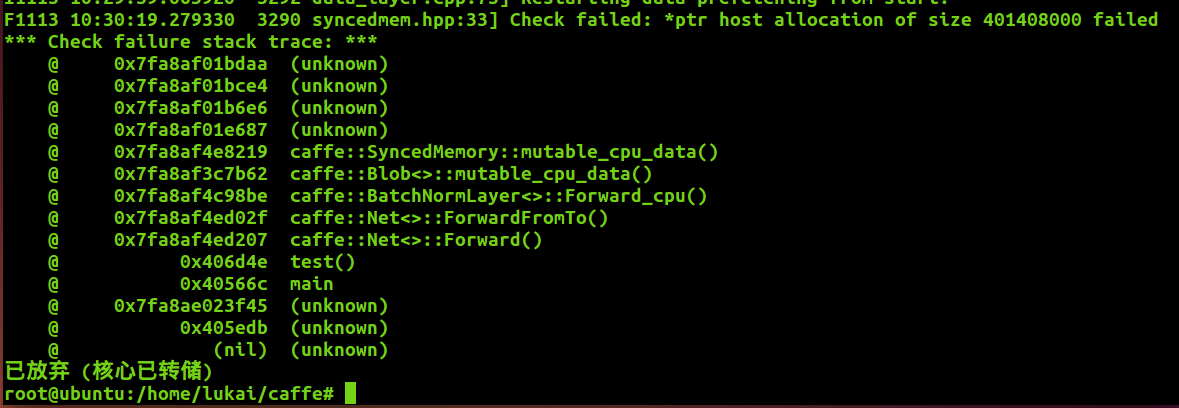
*更新中。。。*

1. 一些问题





GPU内存问题



内存问题

解决方法：将batch\_size改小，但另外的问题是速度。。

慢到难以置信

注意事项：

1.在caffe中运行的所有程序，都必须在根目录下进行，否则会有下面类似的报错：

例子：报not found的错。

2.遇到问题网上相关文章很多，多看，多问，多学。

*更新中。。。*