# MNIST在caffe上的训练与学习

如果前面的train\_net.cpp编译通过了，那么这个就非常简单。

caffe训练和测试的数据都是需要leveldb格式的，niuzhiheng大牛已经给我们转好了MNIST的数据格式。

...\examples\mnist 文件夹下有个get\_mnist\_leveldb.bat ，双击就可以下载到MNIST的leveldb文件。

不能翻墙的，就到：<http://download.csdn.net/detail/u012878523/8140305> 下载吧。

下载完 解压到...\examples\mnist 文件夹下。

网络参数可以再lenet\_train.prototxt中修改。GPU和CPU的切换在lenet\_solver.prototxt中.....

双击train\_lenet。bat就可以训练该网络。

没500轮进行一次测试。

Tset Score 0是正确率。

Test Score 1是测试损失函数值。

# [caffe for windows 训练cifar10](http://blog.csdn.net/u012878523/article/details/41308333)

1 cifar10数据库

60000张32\*32 彩色图片 共10类

50000张训练

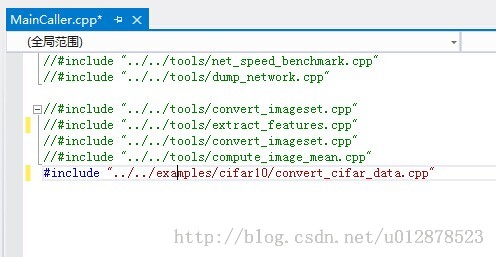
10000张测试

下载cifar10数据库：[http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz](http://www.cs.toronto.edu/~kriz/cifar-10-binary.tar.gz" \t "_blank)

这是binary格式的，所以我们要把它转换成leveldb格式。

2 在../caffe-windows/examples/cifar10文件夹中有一个 convert\_cifar\_data.cpp

将他include到MainCaller.cpp中。如下：



编译....我是一次就通过了 ，在bin文件夹里出现convert\_cifar\_data.exe。然后 就可以进行格式转换。binary→leveldb

可以在bin文件夹下新建一个input文件夹。将cifar10.binary文件放在input文件夹中，这样转换时就不用写路径了。

cmd进入bin文件夹

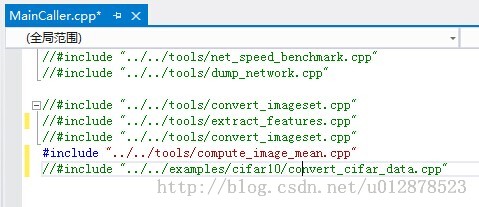


执行后，在output文件夹下有cifar\_train\_leveldb和cifar\_test\_leveldb两个文件夹。里面是转化好的leveldb格式数据。

当然，也可以写一个bat文件处理，方便以后再次使用。

3 下面我们要求数据图像的均值

编译../../tools/comput\_image\_mean.cpp



编译成功后。接下来求mean

cmd进入bin。



执行后，在bin文件夹下出现一个mean.binaryproto文件，这就是所需的均值文件。

4 训练cifar网络

在.../examples/cifar10文件夹里已经有网络的配置文件，我们只需要将cifar\_train\_leveldb和cifar\_test\_leveldb两个文件夹还有mean.binaryproto文件拷到cifar0文件夹下。

修改cifar10\_quick\_train.prototxt中的source: "cifar-train-leveldb"    mean\_file: "mean.binaryproto" 和cifar10\_quick\_test.prototxt中的source: "cifar-test-leveldb"  
    mean\_file: "mean.binaryproto"就可以了，

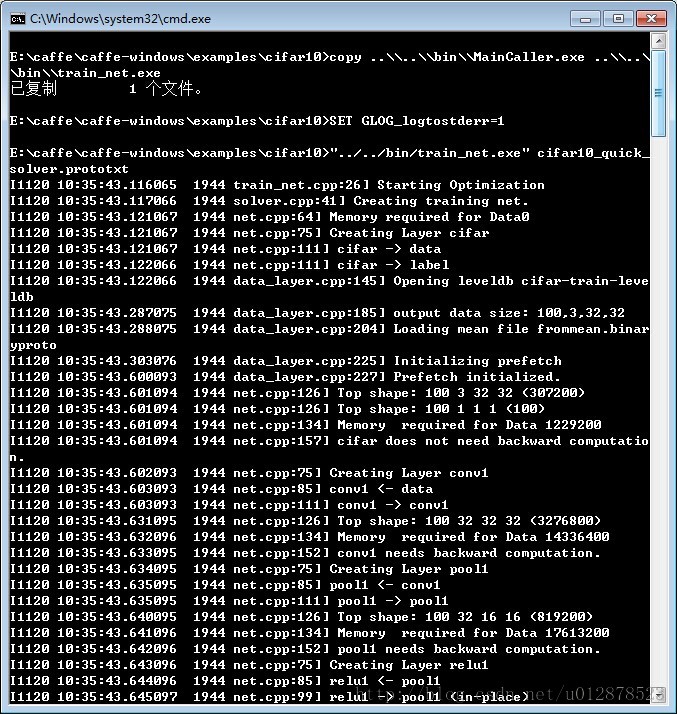
后面再训练就类似于MNIST的训练。写一个train\_quick.bat，内容如下：

**[plain]** [view plaincopy](http://blog.csdn.net/u012878523/article/details/41308333)

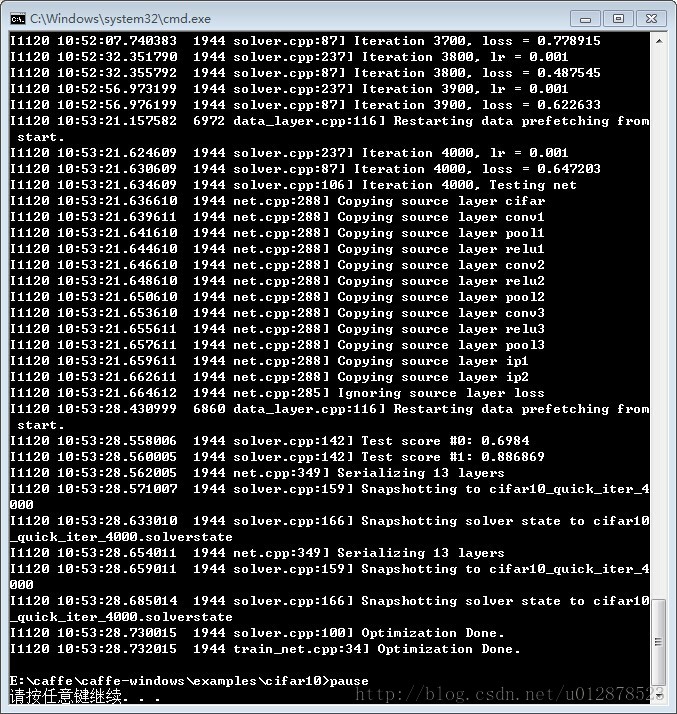
1. copy ..\\..\\bin\\MainCaller.exe ..\\..\\bin\\train\_net.exe
2. SET GLOG\_logtostderr=1
3. "../../bin/train\_net.exe" cifar10\_quick\_solver.prototxt
4. pause

先编译一遍 train\_net.cpp

运行train\_quick.bat



5 运行结果



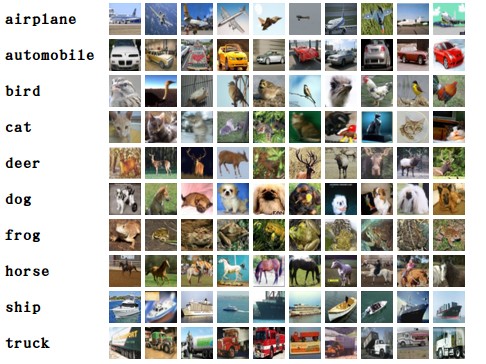
# caffe读书笔记1 CIFAR-10在caffe上进行训练与学习-薛开宇

**1.1使用数据库：CIFAR-10**

60000张 32X32 彩色图像 10类

50000张训练

10000张测试

[](http://photo.blog.sina.com.cn/showpic.html#blogid=5d36d8e00102uya1&url=http://album.sina.com.cn/pic/001HPRtegy6KH14dfqL2f)

**1.2准备**

在终端运行以下指令：

cd $CAFFE\_ROOT/data/cifar10

./get\_cifar10.sh

cd $CAFFE\_ROOT/examples/cifar10

./create\_cifar10.sh

其中CAFFE\_ROOT是caffe-master在你机子的地址

运行之后，将会在examples中出现数据库文件./cifar10-leveldb和数据库图像均值二进制文件./mean.binaryproto

[](http://photo.blog.sina.com.cn/showpic.html#blogid=5d36d8e00102uya1&url=http://album.sina.com.cn/pic/001HPRtegy6KH14fA87ef)

**1.3模型**

该CNN由卷积层，POOLing层，非线性变换层，在顶端的局部对比归一化线性分类器组成。该模型的定义在CAFFE\_ROOT/examples/cifar10 directory’s cifar10\_quick\_train.prototxt中，可以进行修改。其实后缀为prototxt很多都是用来修改配置的。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=5d36d8e00102uya1&url=http://album.sina.com.cn/pic/001HPRtegy6KH14hS6Wde)

**1.4训练和测试**

训练这个模型非常简单，当我们写好参数设置的文件cifar10\_quick\_solver.prototxt和定义的文件cifar10\_quick\_train.prototxt和cifar10\_quick\_test.prototxt后，运行train\_quick.sh或者在终端输入下面的命令：

cd $CAFFE\_ROOT/examples/cifar10

./train\_quick.sh

即可，train\_quick.sh是一个简单的脚本，会把执行的信息显示出来，培训的工具是train\_net.bin,cifar10\_quick\_solver.prototxt作为参数。

然后出现类似以下的信息：

I0317 21:52:48.945710 2008298256 net.cpp:74] Creating Layer conv1

I0317 21:52:48.945716 2008298256 net.cpp:84] conv1 <- data

I0317 21:52:48.945725 2008298256 net.cpp:110] conv1 -> conv1

I0317 21:52:49.298691 2008298256 net.cpp:125] Top shape: 100 32 32 32 (3276800)

I0317 21:52:49.298719 2008298256 net.cpp:151] conv1 needs backward computation.

这是搭建模型的相关信息

接着：

0317 21:52:49.309370 2008298256 net.cpp:166] Network initialization done.

I0317 21:52:49.309376 2008298256 net.cpp:167] Memory required for Data 23790808

I0317 21:52:49.309422 2008298256 solver.cpp:36] Solver scaffolding done.

I0317 21:52:49.309447 2008298256 solver.cpp:47] Solving CIFAR10\_quick\_train

之后，训练开始

I0317 21:53:12.179772 2008298256 solver.cpp:208] Iteration 100, lr = 0.001

I0317 21:53:12.185698 2008298256 solver.cpp:65] Iteration 100, loss = 1.73643

...

I0317 21:54:41.150030 2008298256 solver.cpp:87] Iteration 500, Testing net

I0317 21:54:47.129461 2008298256 solver.cpp:114] Test score #0: 0.5504

I0317 21:54:47.129500 2008298256 solver.cpp:114] Test score #1: 1.27805

其中每100次迭代次数显示一次训练时lr(learning rate),和loss（训练损失函数），每500次测试一次，输出score 0（准确率）

和score 1（测试损失函数）

当5000次迭代之后，正确率约为75%，模型的参数存储在二进制protobuf格式在

cifar10\_quick\_iter\_5000

然后，这个模型就可以用来运行在新数据上了。

**1.5** **其他**

另外，更改cifar\*solver.prototxt文件可以使用CPU训练，

# solver mode: CPU or GPU

solver\_mode: CPU

可以看看CPU和GPU训练的差别。

主要资料来源：caffe官网教程

[caffe读书笔记1 <wbr>CIFAR-10在caffe上进行训练与学习-薛开宇](http://photo.blog.sina.com.cn/showpic.html#blogid=5d36d8e00102uya1&url=http://album.sina.com.cn/pic/001HPRtegy6KH14kjPW20)

# [caffe中cifar10的full\_train模型出现nan的解决办法](http://blog.csdn.net/kuaitoukid/article/details/42120961)

caffe在cifar10 的example上给出了两个模型，一个是quick模式，一个是full模式，训练full模式时会出现loss=nan的错误（当然不会报错，不过出现这个结果就是bug）

自己google了一下，在github上找到了原因跟解决方案，原来是作者把用在cuda-convnet的模型照搬过来了，在caffe上的模型应该去掉LRN层，所以解决的方法很简单：将网络结构中所有的归一化层全部去掉，并修改下一层的bottom参数，然后就不会出现loss=nan的错误了。

当然，如果自己做实验时出现loss=nan的问题时，我的一个解决办法是修改学习率，改的小一点就不会出现错误了。实在不行，就把里面的relu函数变为sigmoid试一试，代价就是训练速度会非常非常慢。

还有一个要检查的点就是要看一下网络的结构是否合理，我在网上下载的network in network的网络结构，最后一层竟然没有一层全连接把输出变为类别数，这让训练陷入了要么出现nan要么结果一直不变的bug境地。

# [在caffe上跑自己的数据](http://blog.csdn.net/u012878523/article/details/41698209)

本文介绍如何使用caffe对自己的图像数据进行分类。

1 图片数据库准备

由于图片数据收集比较费时，为了简单说明，我用了两类，dog和bird，每种约300张。train200张，val100张。

新建一个文件夹mine，放自己的数据，在mine文件夹下新建train和val文件夹，train文件夹下新建bird和dog两个文件夹分别存放200张bird和200张dog，val文件夹里存放其他的图片用于验证。注意，图片的size要归一到相同尺寸。（256\*256）

2 转换成leveldb格式

在mine文件夹下新建两个txt文件：train.txt和val.txt，列出对应图片名及其标签。

数据量较少的可以手动标签，数据量较大的话，可以写批处理命令，比较方便。



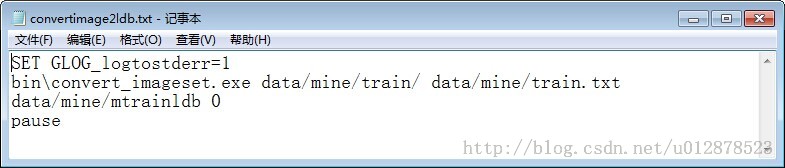
一定要注意，train列表中的图片带相对路径名bird/\*.jpg  dog/\*.jpg。标签1代表bird，标签2代表dog。



生成列表后，编译convert\_imageset.cpp.

在bin文件夹中将刚刚生成的MainCaller.exe重命名为convert\_imageset.exe。做一个批处理命令将图片数据转换成leveldb格式。

在caffe-windows文件夹下新建convertimage2ldb.bat。



双击运行，在mine文件夹下就会出现mtrainldb文件夹。

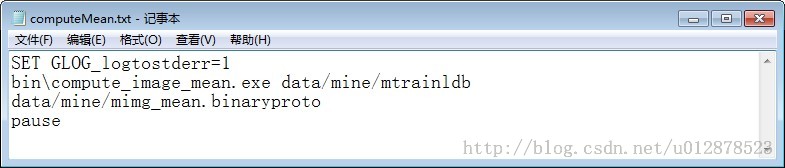
同理可得到mvalldb。这两个就是caffe需要的数据。

注意，我的mine文件夹是放在data文件夹下的，在写convertimage2ldb.bat时注意你自己路径。

3 计算mean

这个比较简单，上篇文章也说了。编译comput\_image\_mean.cpp

在bin文件夹中将刚刚生成的MainCaller.exe重命名为comput\_image\_mean.exe。做一个computeMean.bat方便以后使用。



双击运行之后在mine里面出现mimg\_mean.binaryproto，这就是caffe需要的图片均值文件。

4 训练自己的网络

数据集和均值文件都生成之后，训练和前面两篇文章类似。这次我直接使用的是imagenet的网络结构，几乎没怎么修改，所以我将imagenet里面的imagenet\_train.prototxt、imagenet\_val.prototxt、imagenet\_solver.prototxt直接拷过来修改一下。

imagenet\_val.prototxt、imagenet\_train.prototxt里面的

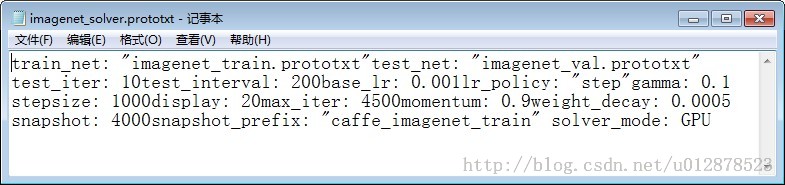
source: "mtrainldb"

mean\_file:"mimg\_mean.binaryproto"

batch\_size: 10

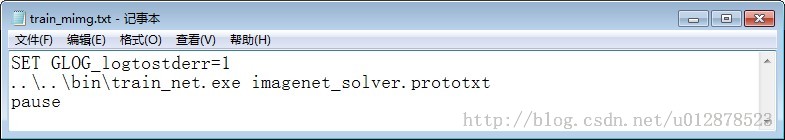
还有最后一层的output改为2，因为我只有两类。

imagenet\_solver.prototxt里面的网络参数修改：



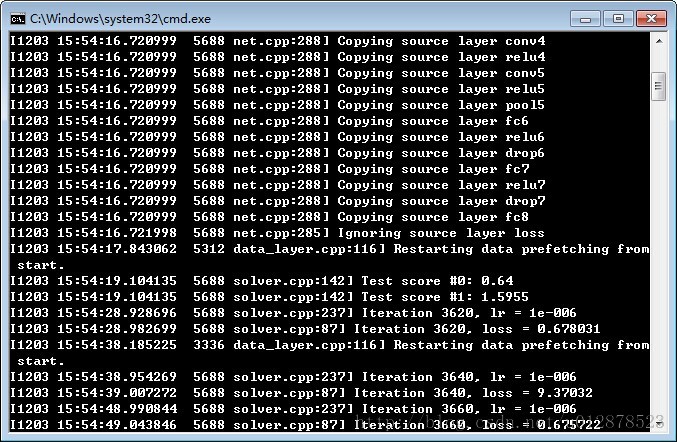
注意最后加上solver\_mode:GPU。

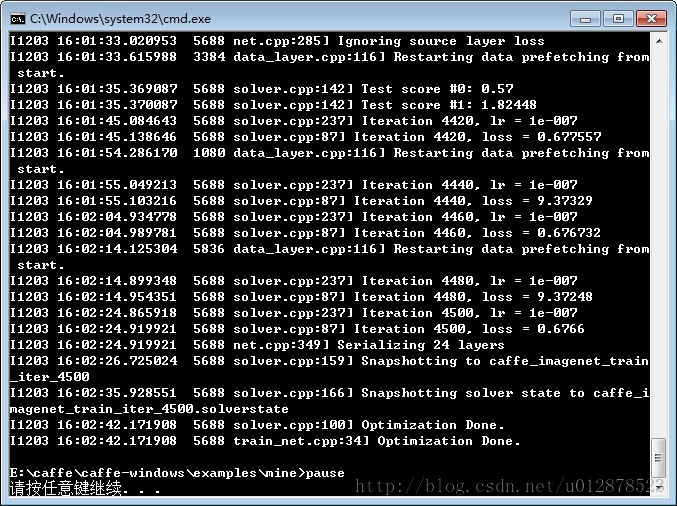
开始训练：



5 实验结果

由于数据量较小，训练比较快。正确率最高能达到0.87。但是最后并不收敛，4500次迭代正确率时高时低。本文只是介绍方法，还有很多参数值得推敲。





# [总结一下用caffe跑图片数据的研究流程](http://blog.csdn.net/lingerlanlan/article/details/31763799)

最近在用caffe玩一些数据集，这些数据集是从淘宝爬下来的图片。主要是想研究一下对女性衣服的分类。

下面是一些具体的操作流程，这里总结一下。

1 爬取数据。写爬虫从淘宝爬取自己需要的数据。

2 数据预处理。将图片从jpg，png格式转为leveldb格式。因为caffe的输入层datalayer是从leveldb读取的。这一步自己基于caffe写了个工具实现转换。

转换命令例子：

./convert\_imagedata.bin /home/linger/imdata/skirt\_train/ /home/linger/linger/testfile/skirt\_train\_db/ /home/linger/linger/testfile/skirt\_train\_attachment/ 3 250 250  
./convert\_imagedata.bin /home/linger/imdata/skirt\_test/ /home/linger/linger/testfile/skirt\_test\_db/ /home/linger/linger/testfile/skirt\_test\_attachment/ 3 250 250

./compute\_image\_mean.bin /home/linger/linger/testfile/skirt\_train\_db /home/linger/linger/testfile/skirt\_train\_mean.binaryproto  
./compute\_image\_mean.bin /home/linger/linger/testfile/skirt\_test\_db /home/linger/linger/testfile/skirt\_test\_mean.binaryproto

3 建立网络模型。主要是写train.prototxt和test.prototxt，还有solver.prototxt。前两者是训练和测试的网络结构，后者是网络的一些配置参数。

4 训练和测试模型。有时候需要调整一些参数，比如learning rate，或者调整网络结构。

5 可视化响应图和权重图。主要把最后一层的特征响应图和权重图画出来，便于观察规律。这两个工具也是自己写的。

工具使用例子：

./visualize\_weights.bin /home/linger/linger/caffe-action/caffe-master/examples/cifar10/cifar10\_full\_test.prototxt /home/linger/linger/caffe-action/caffe-master/examples/cifar10/cifar10\_full\_iter\_60000 1 GPU /home/linger/linger/testfile/skirt\_test\_attachment/image\_filename /home/linger/linger/testfile/weights/ 7  
  
  
./visualize\_features.bin /home/linger/linger/caffe-action/caffe-master/examples/cifar10/cifar10\_full\_test.prototxt /home/linger/linger/caffe-action/caffe-master/examples/cifar10/cifar10\_full\_iter\_60000 20 GPU /home/linger/linger/testfile/skirt\_test\_attachment/image\_filename /home/linger/linger/testfile/innerproduct/ 7

# [[caffe] 数据制作和训练](http://www.cnblogs.com/taokongcn/p/4341371.html)

在使用caffe时，我们希望使用自己的数据进行训练，以下给出如何制作自己的数据。所有的数据制作都是基于imagenet的。

1.数据准备，我们需要一个train和valid文件夹，一个train.txt和val.txt（图片文件夹的位置可以任意，但是这两个txt文件的位置在data/mydata/目录下）

train和valid文件夹自然就存放着待训练的图片，train.txt和val.txt的数据格式如下

[复制代码](javascript:void(0);)

train\_example/1.jpg 0

train\_example/2.jpg 0

train\_example/3.jpg 0

train\_example/4.jpg 0

train\_example/5.jpg 0

train\_example/6.jpg 0

train\_example/7.jpg 0

train\_example/8.jpg 0

train\_example/9.jpg 0

[复制代码](javascript:void(0);)

前边是图片路径，后便是label，注意label是从0开始的。

2. 在example目录下创建自己的文件夹mytask，将imagenet中所有的文件烤过来，然后进行修改。

create\_imagenet.sh

[复制代码](javascript:void(0);)

//当前路径

EXAMPLE=/home/kongtao/caffe-master/examples/voc2007\_test1

//存放train.txt和val.txt的路径，就是data/mydata

DATA=/home/kongtao/caffe-master/data/voc2007\_data1

//不变

TOOLS=/home/kongtao/caffe-master/build/tools

//存放train和val图片文件夹的主目录（这里与txt文件中的目录合起来才是图片目录）

TRAIN\_DATA\_ROOT=/home/kongtao/learn/cnn\_learn/VOC2007\_research/VOCdevkit/

VAL\_DATA\_ROOT=/home/kongtao/learn/cnn\_learn/VOC2007\_research/VOCdevkit/

[复制代码](javascript:void(0);)

运行create\_imagenet.sh，生成两个数据文件lmdb

make\_image\_mean.sh 也是修改相应的路径

最后需要修改的是model中prototxt文件，只需要修改路径

运行

./build/tools/caffe train --solver=models/bvlc\_reference\_caffenet/solver.prototxt

这样就可以了。

以上完成了一个数据文件的创建和训练的过程，对于我们来说，其实网络架构显得更重要，因此不仅仅是调通，更重要的是调整网络架构和改进，这也是最难得部分。

多chanel,需要对数据进行提前白化

[复制代码](javascript:void(0);)

import numpy as np

import caffe

import lmdb

# obviously load your own inputs here -- this is just by way of example

# to show the data can have any channel dimension

# as the dimensions are N x K x H x W

inputs = np.zeros((10, 5, 227, 227))

in\_db = lmdb.open('input-lmdb', map\_size=int(1e12))

with in\_db.begin(write=True) as in\_txn:

for in\_idx, in\_ in enumerate(inputs):

in\_dat = caffe.io.array\_to\_datum(in\_)

in\_txn.put('{:0>10d}'.format(in\_idx), im\_dat.SerializeToString())

in\_db.close()

[复制代码](javascript:void(0);)

记录一下别人制作的东东

http://deepdish.io/2015/04/28/creating-lmdb-in-python/

还有一个

<http://blog.csdn.net/ayst123/article/details/44077903>

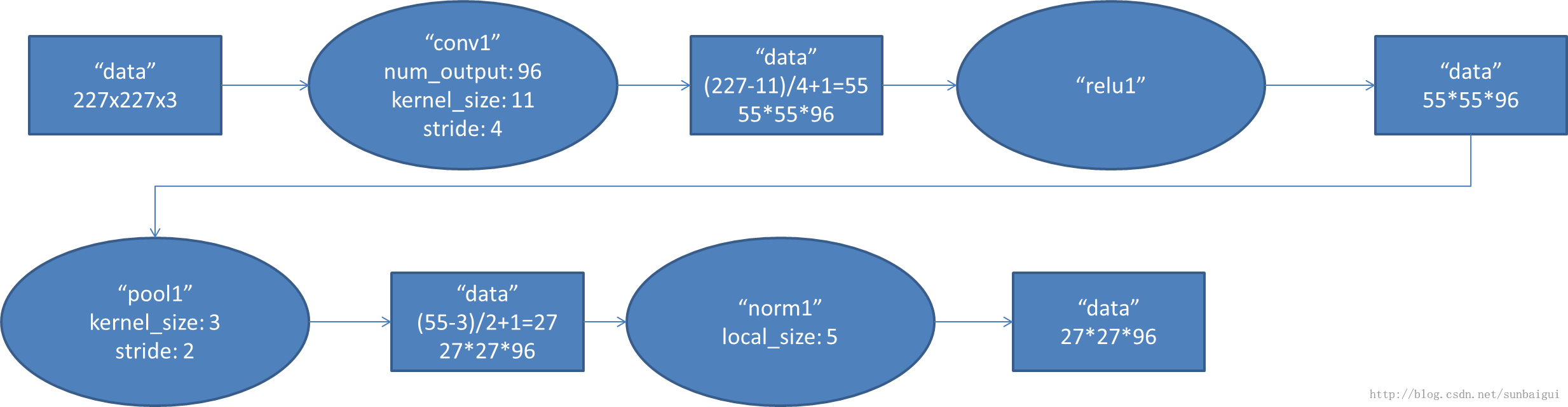
# [[caffe]深度学习之图像分类模型AlexNet解读[caffe]深度学习之图像分类模型AlexNet解读](http://blog.csdn.net/sunbaigui/article/details/39938097)

在imagenet上的图像分类challenge上Alex提出的alexnet网络结构模型赢得了2012届的冠军。要研究CNN类型DL网络模型在图像分类上的应用，就逃不开研究alexnet，这是CNN在图像分类上的经典模型（DL火起来之后）。

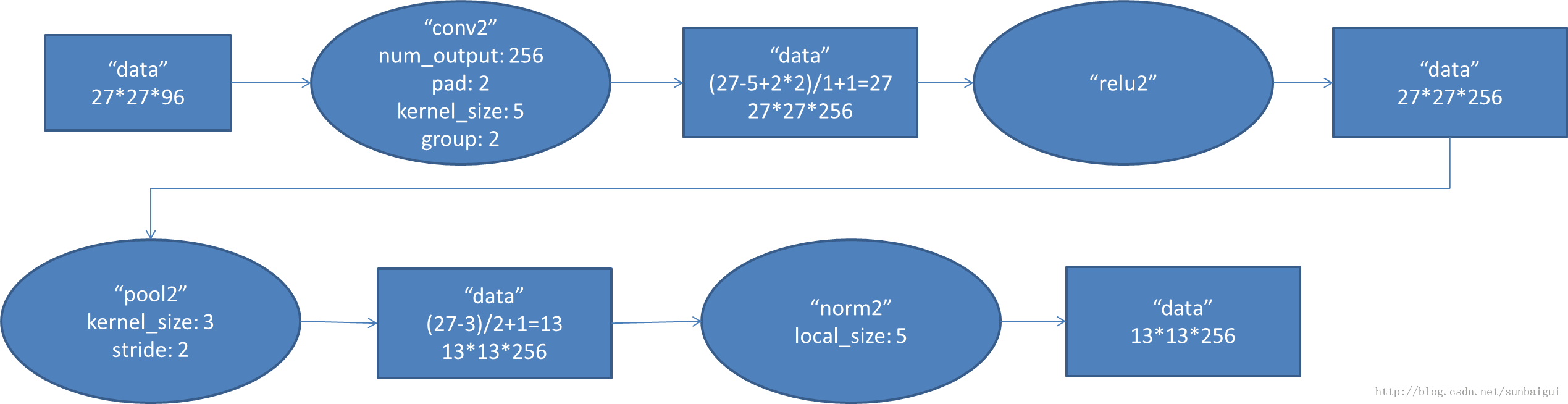
在DL开源实现caffe的model样例中，它也给出了alexnet的复现，具体网络配置文件如下[https://github.com/BVLC/caffe/blob/master/models/bvlc\_reference\_caffenet/train\_val.prototxt](https://github.com/BVLC/caffe/blob/master/models/bvlc_reference_caffenet/train_val.prototxt" \t "_blank)：

**接下来本文将一步步对该网络配置结构中各个层进行详细的解读（训练阶段）：**

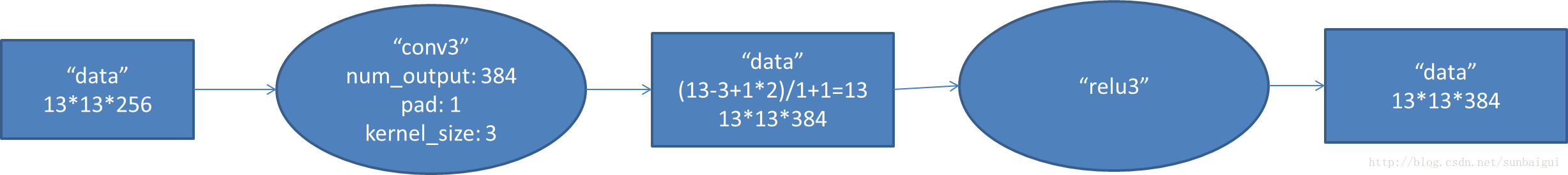
1. conv1阶段DFD（data flow diagram）：



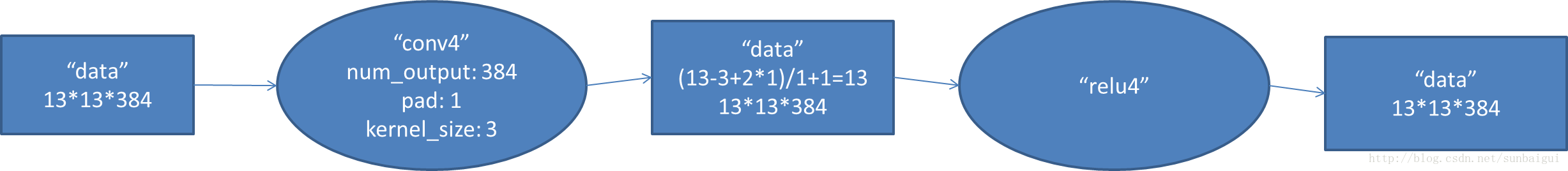
2. conv2阶段DFD（data flow diagram）：



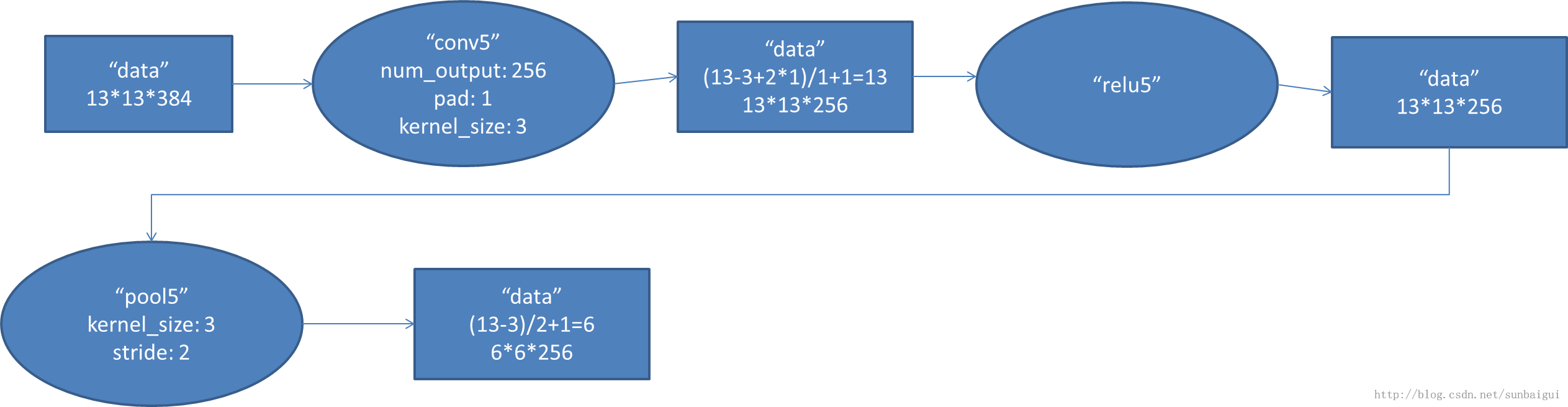
3. conv3阶段DFD（data flow diagram）：



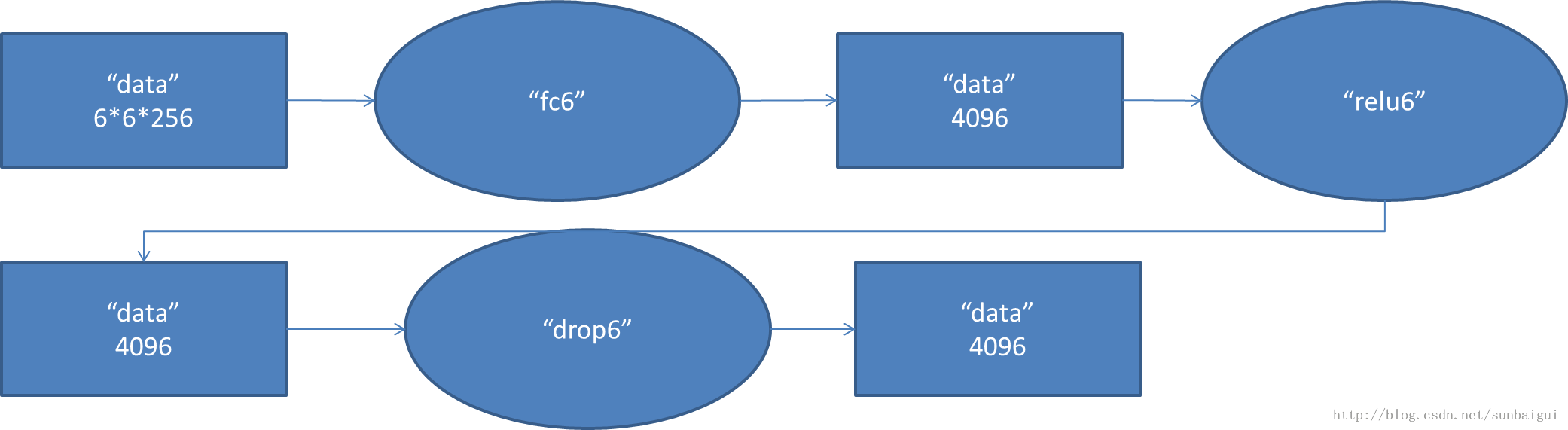
4. conv4阶段DFD（data flow diagram）：



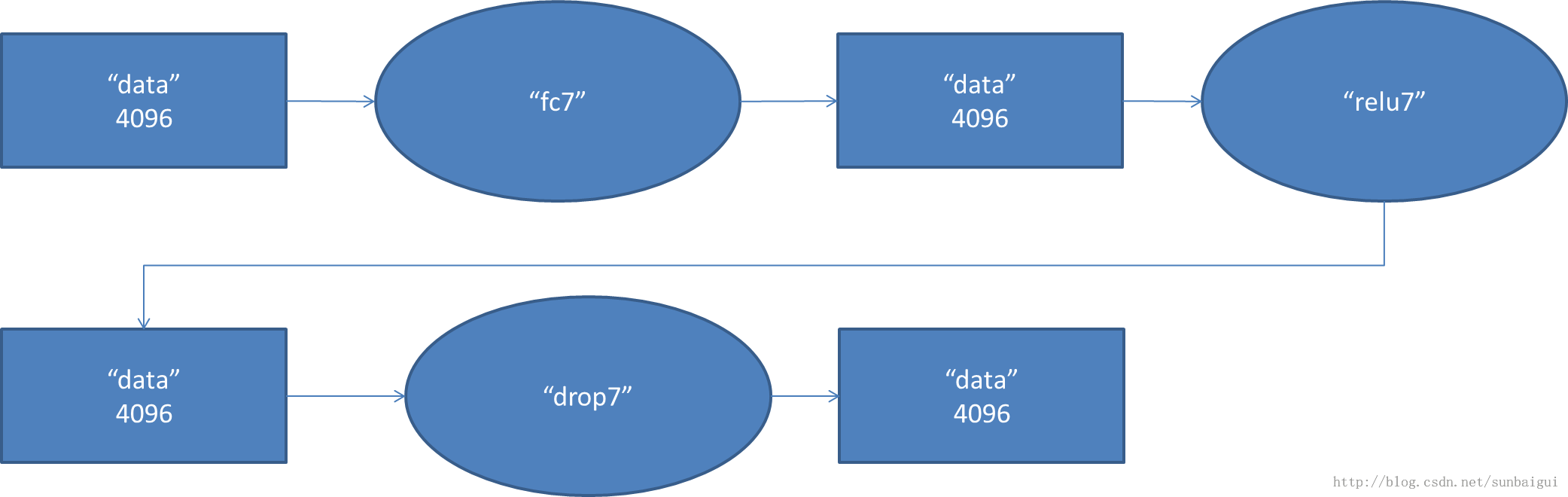
5. conv5阶段DFD（data flow diagram）：



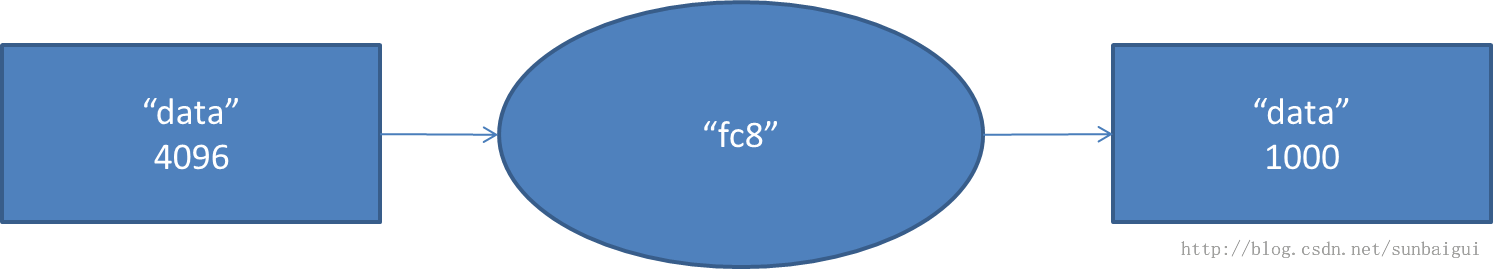
6. fc6阶段DFD（data flow diagram）：



7. fc7阶段DFD（data flow diagram）：



8. fc8阶段DFD（data flow diagram）：



**各种layer的operation更多解释可以参考[http://caffe.berkeleyvision.org/tutorial/layers.html](http://caffe.berkeleyvision.org/tutorial/layers.html" \t "_blank)**

从计算该模型的数据流过程中，该模型参数大概5kw+。

**caffe的输出中也有包含这块的内容日志，详情如下：**

**[cpp]** [view plaincopyprint?](http://blog.csdn.net/sunbaigui/article/details/39938097)[在CODE上查看代码片](https://code.csdn.net/snippets/481505)

1. I0721 10:38:15.326920  4692 net.cpp:125] Top shape: 256 3 227 227 (39574272)
2. I0721 10:38:15.326971  4692 net.cpp:125] Top shape: 256 1 1 1 (256)
3. I0721 10:38:15.326982  4692 net.cpp:156] data does not need backward computation.
4. I0721 10:38:15.327003  4692 net.cpp:74] Creating Layer conv1
5. I0721 10:38:15.327011  4692 net.cpp:84] conv1 <- data
6. I0721 10:38:15.327033  4692 net.cpp:110] conv1 -> conv1
7. I0721 10:38:16.721956  4692 net.cpp:125] Top shape: 256 96 55 55 (74342400)
8. I0721 10:38:16.722030  4692 net.cpp:151] conv1 needs backward computation.
9. I0721 10:38:16.722059  4692 net.cpp:74] Creating Layer relu1
10. I0721 10:38:16.722070  4692 net.cpp:84] relu1 <- conv1
11. I0721 10:38:16.722082  4692 net.cpp:98] relu1 -> conv1 (in-place)
12. I0721 10:38:16.722096  4692 net.cpp:125] Top shape: 256 96 55 55 (74342400)
13. I0721 10:38:16.722105  4692 net.cpp:151] relu1 needs backward computation.
14. I0721 10:38:16.722116  4692 net.cpp:74] Creating Layer pool1
15. I0721 10:38:16.722125  4692 net.cpp:84] pool1 <- conv1
16. I0721 10:38:16.722133  4692 net.cpp:110] pool1 -> pool1
17. I0721 10:38:16.722167  4692 net.cpp:125] Top shape: 256 96 27 27 (17915904)
18. I0721 10:38:16.722187  4692 net.cpp:151] pool1 needs backward computation.
19. I0721 10:38:16.722205  4692 net.cpp:74] Creating Layer norm1
20. I0721 10:38:16.722221  4692 net.cpp:84] norm1 <- pool1
21. I0721 10:38:16.722234  4692 net.cpp:110] norm1 -> norm1
22. I0721 10:38:16.722251  4692 net.cpp:125] Top shape: 256 96 27 27 (17915904)
23. I0721 10:38:16.722260  4692 net.cpp:151] norm1 needs backward computation.
24. I0721 10:38:16.722272  4692 net.cpp:74] Creating Layer conv2
25. I0721 10:38:16.722280  4692 net.cpp:84] conv2 <- norm1
26. I0721 10:38:16.722290  4692 net.cpp:110] conv2 -> conv2
27. I0721 10:38:16.725225  4692 net.cpp:125] Top shape: 256 256 27 27 (47775744)
28. I0721 10:38:16.725242  4692 net.cpp:151] conv2 needs backward computation.
29. I0721 10:38:16.725253  4692 net.cpp:74] Creating Layer relu2
30. I0721 10:38:16.725261  4692 net.cpp:84] relu2 <- conv2
31. I0721 10:38:16.725270  4692 net.cpp:98] relu2 -> conv2 (in-place)
32. I0721 10:38:16.725280  4692 net.cpp:125] Top shape: 256 256 27 27 (47775744)
33. I0721 10:38:16.725288  4692 net.cpp:151] relu2 needs backward computation.
34. I0721 10:38:16.725298  4692 net.cpp:74] Creating Layer pool2
35. I0721 10:38:16.725307  4692 net.cpp:84] pool2 <- conv2
36. I0721 10:38:16.725317  4692 net.cpp:110] pool2 -> pool2
37. I0721 10:38:16.725329  4692 net.cpp:125] Top shape: 256 256 13 13 (11075584)
38. I0721 10:38:16.725338  4692 net.cpp:151] pool2 needs backward computation.
39. I0721 10:38:16.725358  4692 net.cpp:74] Creating Layer norm2
40. I0721 10:38:16.725368  4692 net.cpp:84] norm2 <- pool2
41. I0721 10:38:16.725378  4692 net.cpp:110] norm2 -> norm2
42. I0721 10:38:16.725389  4692 net.cpp:125] Top shape: 256 256 13 13 (11075584)
43. I0721 10:38:16.725399  4692 net.cpp:151] norm2 needs backward computation.
44. I0721 10:38:16.725409  4692 net.cpp:74] Creating Layer conv3
45. I0721 10:38:16.725419  4692 net.cpp:84] conv3 <- norm2
46. I0721 10:38:16.725427  4692 net.cpp:110] conv3 -> conv3
47. I0721 10:38:16.735193  4692 net.cpp:125] Top shape: 256 384 13 13 (16613376)
48. I0721 10:38:16.735213  4692 net.cpp:151] conv3 needs backward computation.
49. I0721 10:38:16.735224  4692 net.cpp:74] Creating Layer relu3
50. I0721 10:38:16.735234  4692 net.cpp:84] relu3 <- conv3
51. I0721 10:38:16.735242  4692 net.cpp:98] relu3 -> conv3 (in-place)
52. I0721 10:38:16.735250  4692 net.cpp:125] Top shape: 256 384 13 13 (16613376)
53. I0721 10:38:16.735258  4692 net.cpp:151] relu3 needs backward computation.
54. I0721 10:38:16.735302  4692 net.cpp:74] Creating Layer conv4
55. I0721 10:38:16.735312  4692 net.cpp:84] conv4 <- conv3
56. I0721 10:38:16.735321  4692 net.cpp:110] conv4 -> conv4
57. I0721 10:38:16.743952  4692 net.cpp:125] Top shape: 256 384 13 13 (16613376)
58. I0721 10:38:16.743988  4692 net.cpp:151] conv4 needs backward computation.
59. I0721 10:38:16.744000  4692 net.cpp:74] Creating Layer relu4
60. I0721 10:38:16.744010  4692 net.cpp:84] relu4 <- conv4
61. I0721 10:38:16.744020  4692 net.cpp:98] relu4 -> conv4 (in-place)
62. I0721 10:38:16.744030  4692 net.cpp:125] Top shape: 256 384 13 13 (16613376)
63. I0721 10:38:16.744038  4692 net.cpp:151] relu4 needs backward computation.
64. I0721 10:38:16.744050  4692 net.cpp:74] Creating Layer conv5
65. I0721 10:38:16.744057  4692 net.cpp:84] conv5 <- conv4
66. I0721 10:38:16.744067  4692 net.cpp:110] conv5 -> conv5
67. I0721 10:38:16.748935  4692 net.cpp:125] Top shape: 256 256 13 13 (11075584)
68. I0721 10:38:16.748955  4692 net.cpp:151] conv5 needs backward computation.
69. I0721 10:38:16.748965  4692 net.cpp:74] Creating Layer relu5
70. I0721 10:38:16.748975  4692 net.cpp:84] relu5 <- conv5
71. I0721 10:38:16.748983  4692 net.cpp:98] relu5 -> conv5 (in-place)
72. I0721 10:38:16.748998  4692 net.cpp:125] Top shape: 256 256 13 13 (11075584)
73. I0721 10:38:16.749011  4692 net.cpp:151] relu5 needs backward computation.
74. I0721 10:38:16.749022  4692 net.cpp:74] Creating Layer pool5
75. I0721 10:38:16.749030  4692 net.cpp:84] pool5 <- conv5
76. I0721 10:38:16.749039  4692 net.cpp:110] pool5 -> pool5
77. I0721 10:38:16.749050  4692 net.cpp:125] Top shape: 256 256 6 6 (2359296)
78. I0721 10:38:16.749058  4692 net.cpp:151] pool5 needs backward computation.
79. I0721 10:38:16.749074  4692 net.cpp:74] Creating Layer fc6
80. I0721 10:38:16.749083  4692 net.cpp:84] fc6 <- pool5
81. I0721 10:38:16.749091  4692 net.cpp:110] fc6 -> fc6
82. I0721 10:38:17.160079  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
83. I0721 10:38:17.160148  4692 net.cpp:151] fc6 needs backward computation.
84. I0721 10:38:17.160166  4692 net.cpp:74] Creating Layer relu6
85. I0721 10:38:17.160177  4692 net.cpp:84] relu6 <- fc6
86. I0721 10:38:17.160190  4692 net.cpp:98] relu6 -> fc6 (in-place)
87. I0721 10:38:17.160202  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
88. I0721 10:38:17.160212  4692 net.cpp:151] relu6 needs backward computation.
89. I0721 10:38:17.160222  4692 net.cpp:74] Creating Layer drop6
90. I0721 10:38:17.160230  4692 net.cpp:84] drop6 <- fc6
91. I0721 10:38:17.160238  4692 net.cpp:98] drop6 -> fc6 (in-place)
92. I0721 10:38:17.160258  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
93. I0721 10:38:17.160265  4692 net.cpp:151] drop6 needs backward computation.
94. I0721 10:38:17.160277  4692 net.cpp:74] Creating Layer fc7
95. I0721 10:38:17.160286  4692 net.cpp:84] fc7 <- fc6
96. I0721 10:38:17.160295  4692 net.cpp:110] fc7 -> fc7
97. I0721 10:38:17.342094  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
98. I0721 10:38:17.342157  4692 net.cpp:151] fc7 needs backward computation.
99. I0721 10:38:17.342175  4692 net.cpp:74] Creating Layer relu7
100. I0721 10:38:17.342185  4692 net.cpp:84] relu7 <- fc7
101. I0721 10:38:17.342198  4692 net.cpp:98] relu7 -> fc7 (in-place)
102. I0721 10:38:17.342208  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
103. I0721 10:38:17.342217  4692 net.cpp:151] relu7 needs backward computation.
104. I0721 10:38:17.342228  4692 net.cpp:74] Creating Layer drop7
105. I0721 10:38:17.342236  4692 net.cpp:84] drop7 <- fc7
106. I0721 10:38:17.342245  4692 net.cpp:98] drop7 -> fc7 (in-place)
107. I0721 10:38:17.342254  4692 net.cpp:125] Top shape: 256 4096 1 1 (1048576)
108. I0721 10:38:17.342262  4692 net.cpp:151] drop7 needs backward computation.
109. I0721 10:38:17.342274  4692 net.cpp:74] Creating Layer fc8
110. I0721 10:38:17.342283  4692 net.cpp:84] fc8 <- fc7
111. I0721 10:38:17.342291  4692 net.cpp:110] fc8 -> fc8
112. I0721 10:38:17.343199  4692 net.cpp:125] Top shape: 256 22 1 1 (5632)
113. I0721 10:38:17.343214  4692 net.cpp:151] fc8 needs backward computation.
114. I0721 10:38:17.343231  4692 net.cpp:74] Creating Layer loss
115. I0721 10:38:17.343240  4692 net.cpp:84] loss <- fc8
116. I0721 10:38:17.343250  4692 net.cpp:84] loss <- label
117. I0721 10:38:17.343264  4692 net.cpp:151] loss needs backward computation.
118. I0721 10:38:17.343305  4692 net.cpp:173] Collecting Learning Rate and Weight Decay.
119. I0721 10:38:17.343327  4692 net.cpp:166] Network initialization done.
120. I0721 10:38:17.343335  4692 net.cpp:167] Memory required **for** Data 1073760256

# [deep learning实践经验总结](http://blog.csdn.net/lingerlanlan/article/details/31773811)

最近拿caffe来做图片分类，遇到不少问题，同时也吸取不少教训和获得不少经验。

先看例子再总结经验。

这是一个2类分类器，分的是条纹衣服和纯色衣服。

先看几张图片。

条纹衣服：

纯色衣服:



肉眼也很容易辨认出来。

训练出来的模型目前的准确率是0.75。

为了可视化特征抽取，我把某一层的特征图和权重图也画出来了，这层是其中一个全连接层。

条纹衣服的特征图：

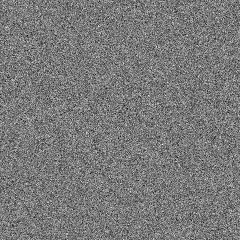
    

纯色衣服的特征图：

看条纹衣服的特征图比较有意思，把“条纹”特征给抽取出来了。也许这就是神经网络神奇的地方，在没有人的干扰的情况下，竟然能学习出来“条纹”特征。

其中一个channel的权重图：



这个就看不出来什么了。曾经有一个数据集，训练的是裙子的模型，当我看到权重图是一个裙子轮廓的图。

好了，说了这么多，总结一下经验吧。

1 数据集要保证质量。曾经玩过一字领和polo领的分类，刚开始效果很差，后来发现有一些“错误”的标签，于是把那些样本给去掉。效果好了很多。

2 learning rate要调整。有一次训练了很久，准确率几乎不变，于是我减少了lr，发现好了很多。

3 均值化图片。实践证明，均值化后再训练收敛速度更快，准确率更高。

对于深度学习的困惑：感觉准确率是个大问题啊，其实这也是所有机器学习算法的通病。

对于别人研究提供的数据集，比如imagenet,cifa10,lenet，效果很好。

但是自己收集的数据集，效果就不是很理想了。

也就是说，算法不是万能的，只是对于某些数据集有效。

我们能做的，是什么？

1 对于哪些数据集，深度学习比较适合？

2 对于效果差的数据集，如何能提高准确率？

曾经脑海里闪过一个念头，是由上面提到的权重图想到的。

当时看到权重图是一个裙子轮廓的图，心里就想。

这是神经网络自动调整出来的权重图，

如果人为加上干预，能否实现优化呢？

# [deep learning实践经验总结2--准确率再次提升，到达0.8，再来总结一下](http://blog.csdn.net/lingerlanlan/article/details/32329761)

近拿caffe来做图片分类，遇到不少问题，同时也吸取不少教训和获得不少经验。

这次拿大摆裙和一步裙做分类，

多次训练效果一直在0.7，后来改动了全链接层的初始化参数。高斯分布的标准差由0.001改为0.0001，就是调小了。

然后效果很明显，准确率高了，权重图画出来后，也看得出是有意义的了，部分权重图是人的轮廓或者裙子的轮廓。

先看看图片：

大摆裙

一步裙

然后找一些响应图看一下，当然我这里展示的是一些效果好的响应图。

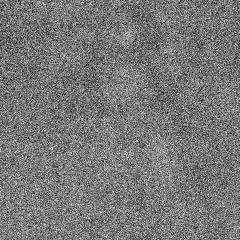
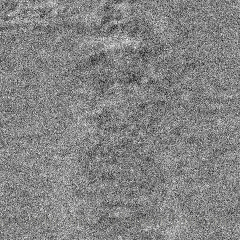
大摆裙

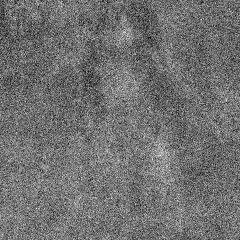
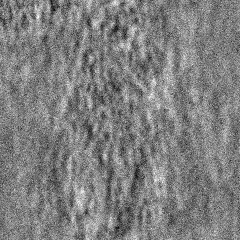
     

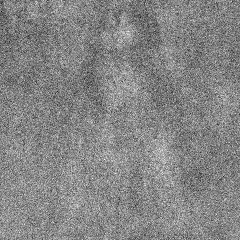
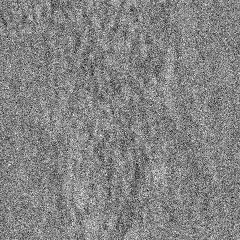
一步裙

一些权重图：

这是网络的结构参数：

**[ruby]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/32329761)

1. name: "CIFAR10\_full\_train"
2. layers {
3. layer {
4. name: "cifar"
5. type: "data"
6. #source: "/home/linger/linger/testfile/crop\_train\_db"
7. #source: "/home/linger/linger/testfile/collar\_train\_db"
8. source: "/home/linger/linger/testfile/skirt\_train\_db"
9. #source: "/home/linger/linger/testfile/pattern\_train\_db"
10. meanfile: "/home/linger/linger/testfile/skirt\_train\_mean.binaryproto"
11. #cropsize: 200
12. batchsize: 20
13. }
14. top: "data"
15. top: "label"
16. }
18. layers {
19. layer {
20. name: "conv1"
21. type: "conv"
22. num\_output: 16
23. kernelsize: 5
24. stride:1
25. weight\_filler {
26. type: "gaussian"
27. std: 0.01
28. }
29. bias\_filler {
30. type: "constant"
31. value: 0.
32. }
33. blobs\_lr: 1.
34. blobs\_lr: 1.
35. weight\_decay: 0.001
36. weight\_decay: 0.
37. }
38. bottom: "data"
39. top: "conv1"
40. }
41. layers {
42. layer {
43. name: "relu1"
44. type: "relu"
45. }
46. bottom: "conv1"
47. top: "conv1"
48. }
49. layers {
50. layer {
51. name: "pool1"
52. type: "pool"
53. pool: MAX
54. kernelsize: 2
55. stride:1
56. }
57. bottom: "conv1"
58. top: "pool1"
59. }
60. layers {
61. layer {
62. name: "conv2"
63. type: "conv"
64. num\_output: 16
65. group: 2
66. kernelsize: 5
67. stride:1
68. weight\_filler {
69. type: "gaussian"
70. std: 0.01
71. }
72. bias\_filler {
73. type: "constant"
74. value: 0.
75. }
76. blobs\_lr: 1.
77. blobs\_lr: 1.
78. weight\_decay: 0.001
79. weight\_decay: 0.
80. }
81. bottom: "pool1"
82. top: "conv2"
83. }
84. layers {
85. layer {
86. name: "relu2"
87. type: "relu"
88. }
89. bottom: "conv2"
90. top: "conv2"
91. }
92. layers {
93. layer {
94. name: "pool2"
95. type: "pool"
96. pool: MAX
97. kernelsize: 2
98. stride: 1
99. }
100. bottom: "conv2"
101. top: "pool2"
102. }
104. layers {
105. layer {
106. name: "ip1"
107. type: "innerproduct"
108. num\_output: 100
109. weight\_filler {
110. type: "gaussian"
111. std: 0.0001
112. }
113. bias\_filler {
114. type: "constant"
115. value: 0.
116. }
117. blobs\_lr: 1.
118. blobs\_lr: 1.
119. weight\_decay: 0.001
120. weight\_decay: 0.
121. }
122. bottom: "pool2"
123. top: "ip1"
124. }
126. layers {
127. layer {
128. name: "ip2"
129. type: "innerproduct"
130. num\_output: 2
131. weight\_filler {
132. type: "gaussian"
133. std: 0.01
134. }
135. bias\_filler {
136. type: "constant"
137. value: 0.
138. }
139. blobs\_lr: 1.
140. blobs\_lr: 1.
141. weight\_decay: 0.001
142. weight\_decay: 0.
143. }
144. bottom: "ip1"
145. top: "ip2"
146. }
148. #-----------------------output------------------------
149. layers {
150. layer {
151. name: "loss"
152. type: "softmax\_loss"
153. }
154. bottom: "ip2"
155. bottom: "label"
156. }

**[ruby]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/32329761)

1. name: "CIFAR10\_full\_test"
2. layers {
3. layer {
4. name: "cifar"
5. type: "data"
6. #source: "/home/linger/linger/testfile/collar\_test\_db"
7. #source: "/home/linger/linger/testfile/crop\_test\_db"
8. source: "/home/linger/linger/testfile/skirt\_test\_db"
9. #source: "/home/linger/linger/testfile/pattern\_test\_db"
10. meanfile: "/home/linger/linger/testfile/skirt\_test\_mean.binaryproto"
11. #cropsize: 200
12. batchsize: 10
13. }
14. top: "data"
15. top: "label"
16. }
18. layers {
19. layer {
20. name: "conv1"
21. type: "conv"
22. num\_output: 16
23. kernelsize: 5
24. stride:1
25. weight\_filler {
26. type: "gaussian"
27. std: 0.01
28. }
29. bias\_filler {
30. type: "constant"
31. value: 0.
32. }
33. blobs\_lr: 1.
34. blobs\_lr: 1.
35. weight\_decay: 0.001
36. weight\_decay: 0.
37. }
38. bottom: "data"
39. top: "conv1"
40. }
41. layers {
42. layer {
43. name: "relu1"
44. type: "relu"
45. }
46. bottom: "conv1"
47. top: "conv1"
48. }
49. layers {
50. layer {
51. name: "pool1"
52. type: "pool"
53. pool: MAX
54. kernelsize: 2
55. stride:1
56. }
57. bottom: "conv1"
58. top: "pool1"
59. }
60. layers {
61. layer {
62. name: "conv2"
63. type: "conv"
64. num\_output: 16
65. group: 2
66. kernelsize: 5
67. stride:1
68. weight\_filler {
69. type: "gaussian"
70. std: 0.01
71. }
72. bias\_filler {
73. type: "constant"
74. value: 0.
75. }
76. blobs\_lr: 1.
77. blobs\_lr: 1.
78. weight\_decay: 0.001
79. weight\_decay: 0.
80. }
81. bottom: "pool1"
82. top: "conv2"
83. }
84. layers {
85. layer {
86. name: "relu2"
87. type: "relu"
88. }
89. bottom: "conv2"
90. top: "conv2"
91. }
92. layers {
93. layer {
94. name: "pool2"
95. type: "pool"
96. pool: MAX
97. kernelsize: 2
98. stride: 1
99. }
100. bottom: "conv2"
101. top: "pool2"
102. }
103. layers {
104. layer {
105. name: "ip1"
106. type: "innerproduct"
107. num\_output: 100
108. weight\_filler {
109. type: "gaussian"
110. std: 0.0001
111. }
112. bias\_filler {
113. type: "constant"
114. value: 0.
115. }
116. blobs\_lr: 1.
117. blobs\_lr: 1.
118. weight\_decay: 0.001
119. weight\_decay: 0.
120. }
121. bottom: "pool2"
122. top: "ip1"
123. }
125. layers {
126. layer {
127. name: "ip2"
128. type: "innerproduct"
129. num\_output: 2
130. weight\_filler {
131. type: "gaussian"
132. std: 0.01
133. }
134. bias\_filler {
135. type: "constant"
136. value: 0.
137. }
138. blobs\_lr: 1.
139. blobs\_lr: 1.
140. weight\_decay: 0.001
141. weight\_decay: 0.
142. }
143. bottom: "ip1"
144. top: "ip2"
145. }
147. #-----------------------output------------------------
148. layers {
149. layer {
150. name: "prob"
151. type: "softmax"
152. }
153. bottom: "ip2"
154. top: "prob"
155. }
156. layers {
157. layer {
158. name: "accuracy"
159. type: "accuracy"
160. }
161. bottom: "prob"
162. bottom: "label"
163. top: "accuracy"
164. }

**[ruby]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/32329761)

1. # reduce learning rate after 120 epochs (60000 iters) by factor 0f 10
2. # then another factor of 10 after 10 more epochs (5000 iters)
4. # The training protocol buffer definition
5. train\_net: "cifar10\_full\_train.prototxt"
6. # The testing protocol buffer definition
7. test\_net: "cifar10\_full\_test.prototxt"
8. # test\_iter specifies how many forward passes the test should carry out.
9. # In the case of CIFAR10, we have test batch size 100 and 100 test iterations,
10. # covering the full 10,000 testing images.
11. test\_iter: 20
12. # Carry out testing every 1000 training iterations.
13. test\_interval: 100
14. # The base learning rate, momentum and the weight decay of the network.
15. base\_lr: 0.00001
16. momentum: 0.9
17. weight\_decay: 0.004
18. # The learning rate policy
19. lr\_policy: "fixed"
20. # Display every 200 iterations
21. display: 20
22. # The maximum number of iterations
23. max\_iter: 60000
24. # snapshot intermediate results
25. snapshot: 1000
26. snapshot\_prefix: "cifar10\_full"
27. # solver mode: 0 for CPU and 1 for GPU
28. solver\_mode: 1

真的是多玩数据，才会对数据形成一种感觉啊。

下次玩3类的。敬请期待！