# caffe源码简单解析——Blob（1）

使用caffe也有一段时间了，但更多是使用Python的接口，使用现有的ImageNet训练好的模型进行图片分类。为了更好的了解caffe这个框架，也为了提高自己的水平，在对卷积神经网络有了一些研究之后，终于开始研读caffe的源码了，今天看了Blob类的一些内容，做个总结。

看过caffe官方文档的话，应该会知道，它可以分为三层：Blob、Layer、Net。Blob是一个四维的数组，用于存储数据，包括输入数据、输出数据、权值等等；Layer层则是神经网络中具体的各层结构，主要是计算的作用，在根据配置文件初始化结构后，前向计算结果，反向更新参数，都是它要做的，而它的输入和输出都是Blob数据；Net的话，就是多个Layer组合而成的有向无环图结构，也就是具体的网络了。Layer和Net的代码有待深入，尤其是Layer的代码，caffe实现了差不多40种不同的Layer层，里面有不同的激活函数，这个要好好研究下。

### Blob源码解析

#include "caffe/common.hpp"

#include "caffe/proto/caffe.pb.h"

#include "caffe/syncedmem.hpp"

#include "caffe/util/math\_functions.hpp"

从blob.hpp包含的四个头文件入手，其中caffe.pb.h是google protocol buffer根据caffe.proto自动生成的，可以到src/caffe/proto/caffe.proto里看下caffe里面用到的各个数据的定义，比如BlobProto，Datum，NetParameter等。使用这个protocol buffer看起来确实方便，一方面可以用文本文件定义结构化的数据类型，另一方面可以生成查询效率更高、占空间更小的二进制文件，具体的教程可以看看[这里](https://developers.google.com/protocol-buffers/docs/tutorials)。

在caffe/common.hpp，主要singleton化Caffe类，并封装了boost和CUDA随机数生成的函数，提供了统一的接口。而在caffe/syncedmem.hpp中，定义了以下的接口：

inline void CaffeMallocHost(void\*\* ptr, size\_t size)

inline void CaffeFreeHost(void\* ptr)

主要是分配内存和释放内存的。而class SyncedMemory定义了内存分配管理和CPU与GPU之间同步的函数，也没啥特别的。

比较重要的是caffe/util/math\_functions.hpp，这里面封装了很多cblas矩阵运算，真是密密麻麻，看的我眼花缭乱、如痴如醉。比如：

void caffe\_cpu\_gemm<float>(const CBLAS\_TRANSPOSE TransA, const CBLAS\_TRANSPOSE TransB, const int M, const int N, const int K, const float alpha, const float\* A, const float\* B, const float beta, float\* C)

封装了cblas\_sgemm(CblasRowMajor, TransA, TransB, M, N, K, alpha, A, lda, B, ldb, beta, C, N)，这个计算得到的结果为C=alphaAB+beta\*C，也即是A和B两个矩阵的乘积。[这里](https://developer.apple.com/library/IOs/documentation/Accelerate/Reference/BLAS_Ref/index.html#//apple_ref/c/func/cblas_sgemm)有详细的解释。

void caffe\_cpu\_gemv<float>(const CBLAS\_TRANSPOSE TransA, const int M, const int N, const float alpha, const float\* A, const float\* x, const float beta, float\* y)

是对cblas\_sgemv的封装，实现的矩阵与向量的乘积，结果为y=alphaAx+beta\*y。

void caffe\_axpy<float>(const int N, const float alpha, const float\* X, float\* Y)

封装了cblas\_saxpy，实现的是Y=alpha\*X+Y

里面都是诸如此类的函数，基本是些矩阵和向量的一些处理函数。

回到blob类，里面定义了data\_(),diff\_()指针，用于存放数据，而num\_, channel\_, height\_, width\_则主要用来做定位offset和reshape处理。对于输入(n, c, h, w)位置的数据位置为((n\*channels\_+c)\*height\_+h)\*width\_+w，可以依据位置取data\_()或diff\_()中的数据。

对blob的理解还要结合caffe.proto里面BlobProto的定义：

message BlobProto {

optional int32 num = 1 [default = 0];

optional int32 channels = 2 [default = 0];

optional int32 height = 3 [default = 0];

optional int32 width = 4 [default = 0];

repeated float data = 5 [packed = true];

repeated float diff = 6 [packed = true];

}

对于BlobProto，可以看到定义了四个optional的int32类型的名字（name）num、channels、height和width，optional意味着Blob可以有一个或者没有这个参数，每个名字（name）后面都有一个数字，这个数字是其名字的一个**标签**。这个数字就是用来在生成的二进制文件中搜索查询的标签（怪不得会快呢^\_^）。关于这个数字，1到15会花费1byte的编码空间，16到2047花费2byte。所以一般建议把那些频繁使用的名字的标签设为1到15之间的值～而后面的repeated意味着float类型的data和diff可以重复任意次，而加上[packed = true]是为了更高效的编码。

到这里基本上Blob就很清楚了，主要数据有两个data和diff，用num、channels、height和width这四个维度来确定数据的具体位置，做一些数据查询和Blobreshape的操作。

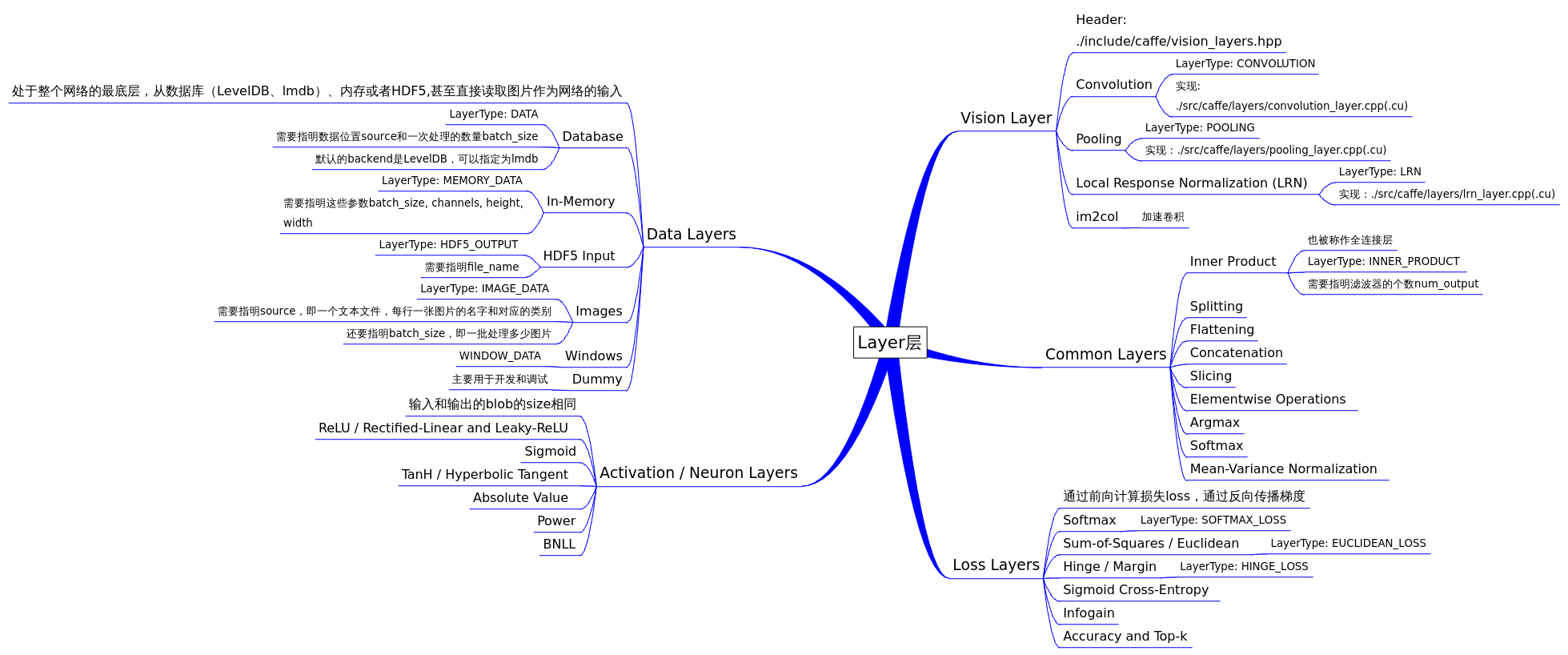
关于Blob就这么多内容，毕竟就是一个统一的数据存取接口，后续会重点读一下Layer的源码，毕竟各层的输入输出和计算更新过程都在里面，还需要补充一些相关的知识～～

目前的感受，是学到了一些封装的手法，可以看看封装cblas函数的那个文件，以及CPU和GPU一些接口的封装上；另一方面是对于Protocol Buffer有了一些了解，目前看起来确实方便，以后如果遇到类似的场景可以试着用一下～～

# caffe源码简单解析——Layer层

### 前言

老实说，caffe中的layer层代码比较多，各种抽象看起来比较绕。官方关于[Layer的教程](http://caffe.berkeleyvision.org/tutorial/layers.html)写的很清楚，我根据这个文档，简单画了个图，再理解起来就方便了一些。



### layer.hpp

和layer相关的头文件有：

common\_layers.hpp

data\_layers.hpp

layer.hpp

loss\_layers.hpp

neuron\_layers.hpp

vision\_layers.hpp

其中``layer.hpp是抽象出来的基类，其他都是在其基础上的继承，也即剩下的五个头文件和上图中的五个部分。在layer.hpp`头文件里，包含了这几个头文件：

#include "caffe/blob.hpp"

#include "caffe/common.hpp"

#include "caffe/proto/caffe.pb.h"

#include "caffe/util/device\_alternate.hpp"

在device\_alternate.hpp中，通过#ifdef CPU\_ONLY定义了一些宏来取消GPU的调用：

#define STUB\_GPU(classname)

#define STUB\_GPU\_FORWARD(classname, funcname)

#define STUB\_GPU\_BACKWARD(classname, funcname)

layer中有这三个主要参数：

LayerParameter layer\_param\_; // 这个是protobuf文件中存储的layer参数

vector<share\_ptr<Blob<Dtype>>> blobs\_; // 这个存储的是layer的参数，在程序中用的

vector<bool> param\_propagate\_down\_; // 这个bool表示是否计算各个blob参数的diff，即传播误差

Layer类的构建函数explicit Layer(const LayerParameter& param) : layer\_param\_(param)会尝试从protobuf文件读取参数。其三个主要接口：

virtual void SetUp(const vector<Blob<Dtype>\*>& bottom, vector<Blob<Dtype>\*>\* top)

inline Dtype Forward(const vector<Blob<Dtype>\*>& bottom, vector<Blob<Dtype>\*>\* top);

inline void Backward(const vector<Blob<Dtype>\*>& top, const vector<bool>& propagate\_down, const <Blob<Dtype>\*>\* bottom);

SetUp函数需要根据实际的参数设置进行实现，对各种类型的参数初始化；Forward和Backward对应前向计算和反向更新，输入统一都是bottom，输出为top，其中Backward里面有个propagate\_down参数，用来表示该Layer是否反向传播参数。

在Forward和Backward的具体实现里，会根据Caffe::mode()进行对应的操作，即使用cpu或者gpu进行计算，两个都实现了对应的接口Forward\_cpu、Forward\_gpu和Backward\_cpu、Backward\_gpu，这些接口都是virtual，具体还是要根据layer的类型进行对应的计算（注意：有些layer并没有GPU计算的实现，所以封装时加入了CPU的计算作为后备）。另外，还实现了ToProto的接口，将Layer的参数写入到protocol buffer文件中。

### data\_layers.hpp

data\_layers.hpp这个头文件包含了这几个头文件：

#include "boost/scoped\_ptr.hpp"

#include "hdf5.h"

#include "leveldb/db.h"

#include "lmdb.h"

#include "caffe/blob.hpp"

#include "caffe/common.hpp"

#include "caffe/filler.hpp"

#include "caffe/internal\_thread.hpp"

#include "caffe/layer.hpp"

#include "caffe/proto/caffe.pb.h"

看到hdf5、leveldb、lmdb，确实是与具体数据相关了。data\_layer作为原始数据的输入层，处于整个网络的最底层，它可以从数据库leveldb、lmdb中读取数据，也可以直接从内存中读取，还可以从hdf5，甚至是原始的图像读入数据。

关于这几个数据库，简介如下：

LevelDB是Google公司搞的一个高性能的key/value存储库，调用简单，数据是被Snappy压缩，据说效率很多，可以减少磁盘I/O，具体例子可以看看[维基百科](http://zh.wikipedia.org/wiki/LevelDB)。

而[LMDB](http://symas.com/mdb/)（Lightning Memory-Mapped Database），是个和levelDB类似的key/value存储库，但效果似乎更好些，其首页上写道“ultra-fast，ultra-compact”，这个有待进一步学习啊～～

HDF（Hierarchical Data Format）是一种为存储和处理大容量科学数据而设计的文件格式及相应的库文件，当前最流行的版本是HDF5,其文件包含两种基本数据对象：

* 群组（group）：类似文件夹，可以包含多个数据集或下级群组；
* 数据集（dataset）：数据内容，可以是多维数组，也可以是更复杂的数据类型。

以上内容来自[维基百科](http://zh.wikipedia.org/wiki/HDF)，关于使用可以参考[HDF5 小试——高大上的多对象文件格式](HDF5 小试——高大上的多对象文件格式)，后续会再详细的研究下怎么用。

caffe/filler.hpp的作用是在网络初始化时，根据layer的定义进行初始参数的填充，下面的代码很直观，根据FillerParameter指定的类型进行对应的参数填充。

// A function to get a specific filler from the specification given in

// FillerParameter. Ideally this would be replaced by a factory pattern,

// but we will leave it this way for now.

template <typename Dtype>

Filler<Dtype>\* GetFiller(const FillerParameter& param) {

const std::string& type = param.type();

if (type == "constant") {

return new ConstantFiller<Dtype>(param);

} else if (type == "gaussian") {

return new GaussianFiller<Dtype>(param);

} else if (type == "positive\_unitball") {

return new PositiveUnitballFiller<Dtype>(param);

} else if (type == "uniform") {

return new UniformFiller<Dtype>(param);

} else if (type == "xavier") {

return new XavierFiller<Dtype>(param);

} else {

CHECK(false) << "Unknown filler name: " << param.type();

}

return (Filler<Dtype>\*)(NULL);

}

internal\_thread.hpp里面封装了pthread函数，继承的子类可以得到一个单独的线程，主要作用是在计算当前的一批数据时，在后台获取新一批的数据。

关于data\_layer，基本要注意的我都在图片上标注了。

### neuron\_layers.hpp

输入了data后，就要计算了，比如常见的sigmoid、tanh等等，这些都计算操作被抽象成了neuron\_layers.hpp里面的类NeuronLayer，这个层只负责具体的计算，因此明确定义了输入ExactNumBottomBlobs()和ExactNumTopBlobs()都是常量1,即输入一个blob，输出一个blob。

### common\_layers.hpp

NeruonLayer仅仅负责简单的一对一计算，而剩下的那些复杂的计算则通通放在了common\_layers.hpp中。像ArgMaxLayer、ConcatLayer、FlattenLayer、SoftmaxLayer、SplitLayer和SliceLayer等各种对blob增减修改的操作。

### loss\_layers.hpp

前面的data layer和common layer都是中间计算层，虽然会涉及到反向传播，但传播的源头来自于loss\_layer，即网络的最终端。这一层因为要计算误差，所以输入都是2个blob，输出1个blob。

### vision\_layers.hpp

vision\_layer主要是图像卷积的操作，像convolusion、pooling、LRN都在里面，按官方文档的说法，是可以输出图像的，这个要看具体实现代码了。里面有个im2col的实现，看caffe作者的解释，主要是为了加速卷积的，这个具体是怎么实现的要好好研究下～～

### 后语

结合官方文档，再加画图和看代码，终于对整个layer层有了个基本认识：data负责输入，vision负责卷积相关的计算，neuron和common负责中间部分的数据计算，而loss是最后一部分，负责计算反向传播的误差。具体的实现都在src/caffe/layers里面，慢慢再研究研究。

在这些抽象的基类头文件里，看起来挺累，好在各种搜索，也能学到一些技巧，如，[巧用宏定义来简写C，C++代码](http://www.vimer.cn/2011/01/%E5%B7%A7%E7%94%A8%E5%AE%8F%E5%AE%9A%E4%B9%89%E6%9D%A5%E7%AE%80%E5%86%99%E4%BB%A3%E7%A0%81.html)，使用模板方法，将有大量重复接口和参数的类抽象为一个宏定义，达到简化代码的目的。

# 基于caffe的特征可视化

最近想看一看卷积神经网络中各层的卷积结果，但在网上搜索feature visualization并没能找到通俗易懂的内容。在caffe的官网教程中，有这么一个[Instant Recognition with Caffe](http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/00-classification.ipynb)，参照它可以很快的做出可视化的结果。但感觉里面还是稍有些复杂，这里做了些简化，达到目的即可。

首先，import相关的包：

import caffe

import matplotlib.pyplot as plt

import matplotlib.cm as cm

import numpy as np

利用配置文件载入训练好的模型：

net = caffe.Classifier("./alexnet\_deploy.prototxt","bvlc\_alexnet.caffemodel", mean=np.float([104.0, 116.0, 122.0]), channel\_swap=(2,1,0))

这里我使用的是alexnet模型。

caffe中的数据都是在blobs中存的，我们可以看一下其结构：

In [103]: net.blobs

Out[103]: OrderedDict([('data', <caffe.\_caffe.Blob object at 0x7f9831f36aa0>), ('conv1', <caffe.\_caffe.Blob object at 0x7f9831f369b0>), ('norm1', <caffe.\_caffe.Blob object at 0x7f9831f36e60>), ('pool1', <caffe.\_caffe.Blob object at 0x7f9831f36c80>), ('conv2', <caffe.\_caffe.Blob object at 0x7f9831f36c08>), ('norm2', <caffe.\_caffe.Blob object at 0x7f9831f36f50>), ('pool2', <caffe.\_caffe.Blob object at 0x7f9832080a28>), ('conv3', <caffe.\_caffe.Blob object at 0x7f98320809b0>), ('conv4', <caffe.\_caffe.Blob object at 0x7f9832080d70>), ('conv5', <caffe.\_caffe.Blob object at 0x7f9832080aa0>), ('pool5', <caffe.\_caffe.Blob object at 0x7f9832080848>), ('fc6', <caffe.\_caffe.Blob object at 0x7f9832080b90>), ('fc7', <caffe.\_caffe.Blob object at 0x7f9832080e60>), ('fc8', <caffe.\_caffe.Blob object at 0x7f98320807d0>), ('prob', <caffe.\_caffe.Blob object at 0x7f9832080de8>)])

可以看到输入的data，五层卷积+pooling，以及三层全连接fc，和最后的预测概率结果prob。

然后，使用这个模型做预测：

net.predict([caffe.io.load\_image("./image.jpg")])

这里会直接输出1000维的数组，每个值都是预测该类的概率。这预测结果不是重点，可以忽略。

跑了一遍预测之后，各层就有了具体的数据，我们就可以看每层的可视化结果了。比如，conv1层：

In [104]: net.blobs['conv1']

Out[104]: <caffe.\_caffe.Blob at 0x7f9832080b18>

In [105]: conv1 = net.blobs['conv1']

In [106]: conv1.

conv1.channels conv1.data conv1.height conv1.reshape

conv1.count conv1.diff conv1.num conv1.width

可以看到conv1层有8个参数，其中channels是该层的卷积核的数量，即96, height和width分别是该层结果的高和宽，即55×55, num是patch的个数，是在模型的配置文件中alexnet\_deploy.prototxt设置好的，这里是10, count是参数的总个数，本层有2904000个参数，diff是训练时使用的误差，这里不是训练，所以全部为0,data里包含了我们需要的数据，其维度为10×96×55×55，因为我们就识别了一张图片，所以只取第一个就可以了：

conv1\_data = conv1.data[0,:]

要可视化就需要显示结果，可以使用下面的函数：

def vis\_square(data, padsize=1, padval=0):

data -= data.min()

data /= data.max()

n = int(np.ceil(np.sqrt(data.shape[0])))

padding = ((0, n\*\*2-data.shape[0]), (0, padsize), (0, padsize))+((0,0),)\*(data.ndim-3)

data = np.pad(data, padding, mode='constant', constant\_values=(padval, padval))

data = data.reshape((n,n)+data.shape[1:]).transpose((0,2,1,3)+tuple(range(4, data.ndim+1)))

data = data.reshape((n\*data.shape[1], n\*data.shape[3])+data.shape[4:])

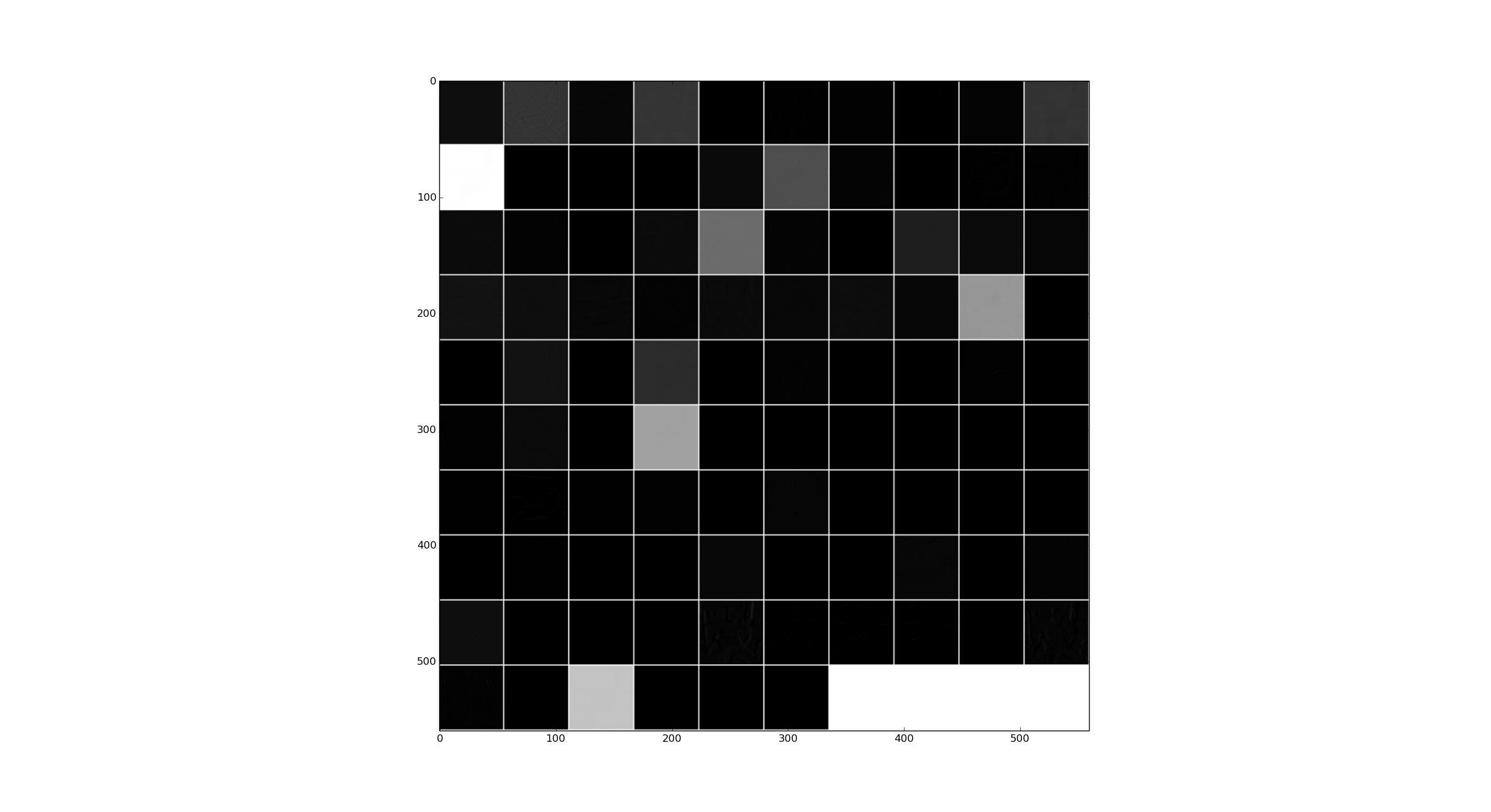
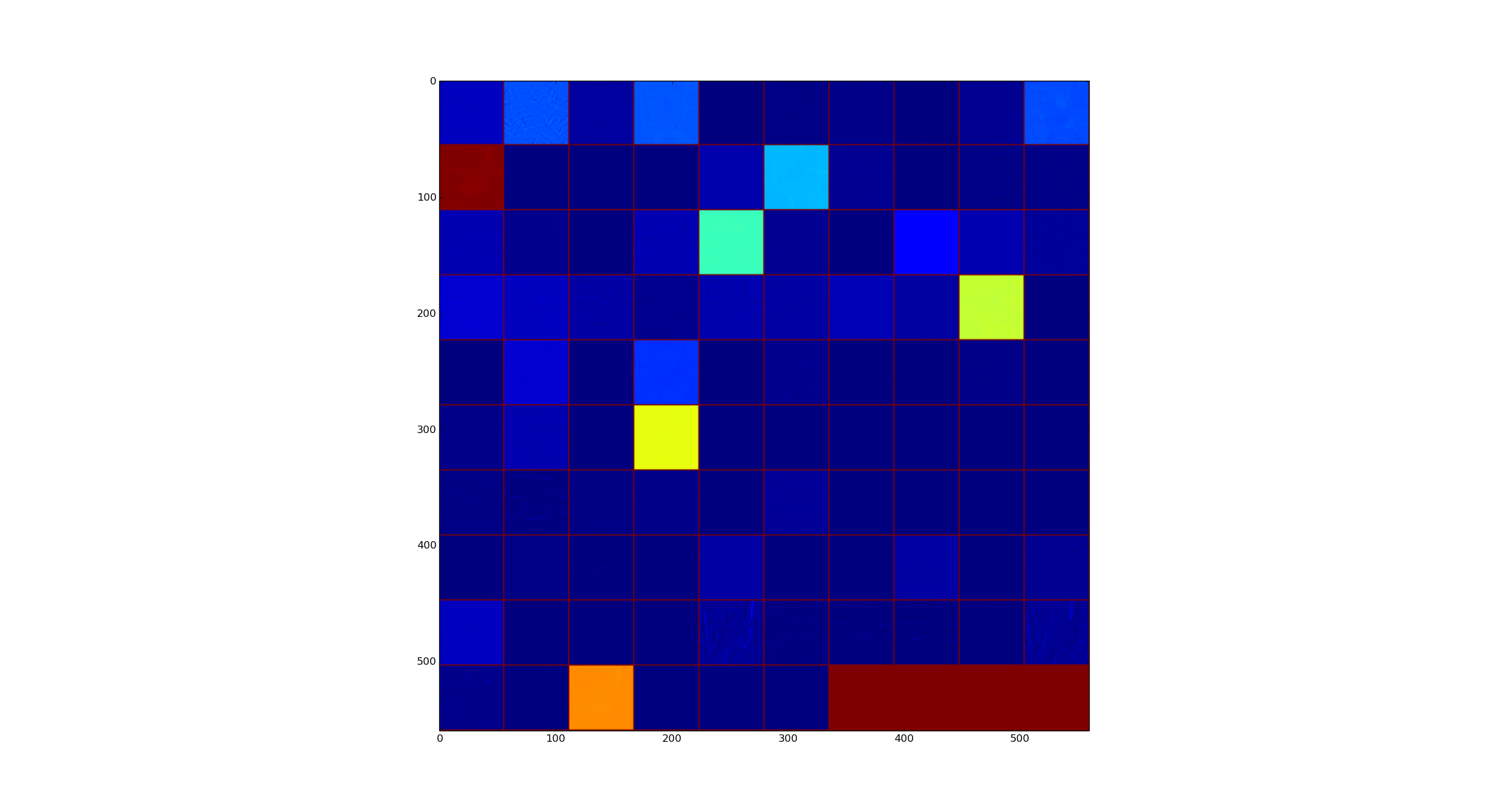
plt.imshow(data, cmap=cm.gray)

这个函数来自于caffe官方的文档，我仅在最后一行做了修改，添加了cmap=cm.gray，让结果显示为灰度，而不是彩色的。

先看下卷积第一层96个结果：

In [122]: vis\_square(conv1\_data, padval=1)

In [123]: plt.show()

结果如下，看灰度可能不太明显，可以删掉函数vis\_square中最后一行的cmap=cm.gray，得到彩色的结果，看起来就明显多了。  
  


看下第85个结果：

In [180]: res = conv1\_data[84:85, :]

In [181]: vis\_square(res, padval=1)

In [182]: plt.show()

