**caffe中学习率调整策略lr\_policy的定义：**

 /caffe-master/src/caffe/proto/caffe.proto file 中有各种学习率调整策略的定义：

// The learning rate decay policy. The currently implemented learning rate

// policies are as follows:

// - fixed: always return base\_lr.

// - step: return base\_lr \* gamma ^ (floor(iter / step))

// - exp: return base\_lr \* gamma ^ iter

// - inv: return base\_lr \* (1 + gamma \* iter) ^ (- power)

// - multistep: similar to step but it allows non uniform steps defined by

// stepvalue

// - poly: the effective learning rate follows a polynomial decay, to be

// zero by the max\_iter. return base\_lr (1 - iter/max\_iter) ^ (power)

// - sigmoid: the effective learning rate follows a sigmod decay

// return base\_lr ( 1/(1 + exp(-gamma \* (iter - stepsize))))

**caffe提供六种优化算法：**

Stochastic Gradient Descent (type: "SGD"),

AdaDelta (type: "AdaDelta"),

Adaptive Gradient (type: "AdaGrad"),

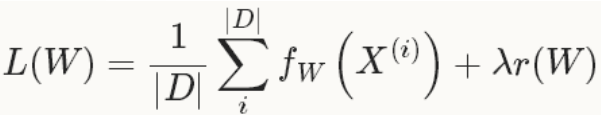
Adam (type: "Adam"),

Nesterov’s Accelerated Gradient (type: "Nesterov") and

RMSprop (type: "RMSProp")

**caffe的SGD优化算法：**

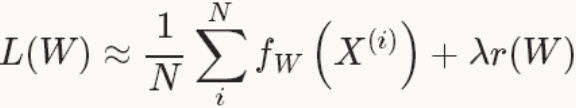
对于一个数据集D，需要优化的目标函数是整个数据集中所有数据loss的平均值：



其中，fW(x(i))计算的是数据样本x(i)上的loss, 先将每个单独的样本x的loss求出来，然后求和，最后求均值。 r(W)是正则项。

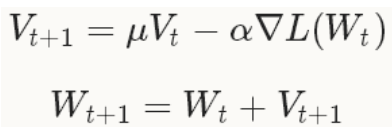
如果采用这种Loss 函数，迭代一次需要计算整个数据集，在数据集非常大|D|的这情况下，效率很低，这个也是我们熟知的梯度下降采用的方法。

在实际中，通过将整个数据集分成几批（batches), 每一批就是一个mini-batch，其数量（batch\_size)为N<<|D|，此时的loss 函数为：



每利用一个mini-batch计算一次Loss，就反向传播更新一次梯度，效率高

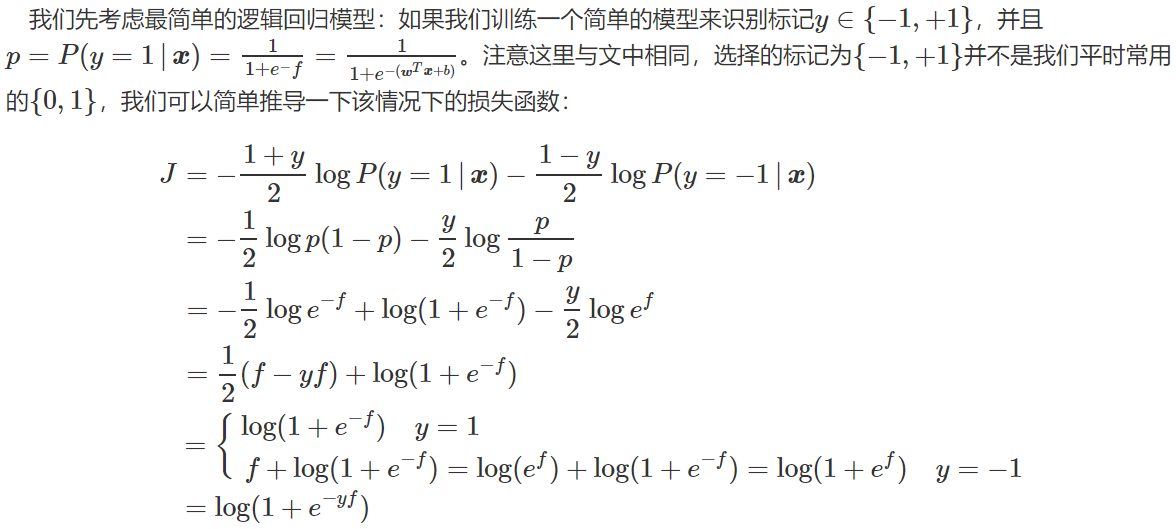
随机梯度下降（SGD，Stochastic gradient descent）是在梯度下降法（gradient descent）的基础上发展起来的。SGD通过负梯度和上一次的权重更新值Vt的线性组合来更新W，迭代公式如下：



其中，α是负梯度的学习率(由base\_lr+学习率调整策略而来)

μ是上一次梯度值的权重（即momentum），用来加权之前梯度方向对现在梯度下降方向的影响。这两个参数需要通过tuning来得到最好的结果，一般是根据经验设定的。

**二分类模型的交叉熵损失函数：**

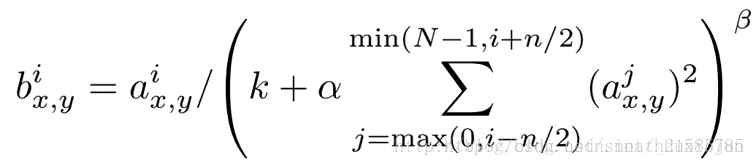


**LRN** **(Local Response Normalization)：**

在2015年 Very Deep Convolutional Networks for Large-Scale Image Recognition证明LRN基本没什么用

对局部神经元的活动创建竞争机制，使得其中响应比较大的值变得相对更大，并抑制其他反馈较小的神经元，**增强了模型的泛化能力**：

局部响应归一化原理是仿造生物学上活跃的神经元对相邻神经元的抑制现象（侧抑制），然后根据论文有公式如下



这个公式中的a表示卷积层（包括卷积操作和池化操作）后的输出结果，这个输出结果的结构是一个四维数组[batch, height, width, channel]。ai(x,y)表示在这个输出结构中的一个位置[a,b,c,d]，即第a张图的第i个通道下的高度为x宽度为y的点。论文公式中的N表示通道数(channel)。a,n/2,k,α,β分别表示函数中的input,depth\_radius,bias,alpha,beta，其中n/2,k,α,β都是自定义的，特别注意一下∑叠加的方向是沿着通道方向的

万能逼近定理保证一个具有至少一个隐藏层的神经网络可以以任意精度表示任意函数，只要其隐藏层被允许有足够多的神经单元。

* Sigmoid -> Tanh -> ReLU -> Leaky ReLU -> Maxout：一个神经元有一个激活函数
* Softmax是一个特殊的激活函数，只会被用在网络中的最后一层，最后一层共用一个Softmax函数

Keras搭建一个网络;

from keras import models

from keras import layers

network = models.Sequential()

network.add(layers.Dense(512, activation='relu', input\_shape=(28 \* 28,)))

network.add(layers.Dense(10, activation='softmax'))

lmdb数据格式常用于单标签数据，像分类等，经常使用lmdb的数据格式。对于回归等问题，或者多标签数据，一般使用h5py数据的格式

Caffe生成的数据分为2种格式：Lmdb和Leveldb。lmdb的速度比leveldb快10%至15%，更重要的是lmdb允许多种训练模型同时读取同一组数据集。因此lmdb取代了leveldb成为Caffe默认的数据集生成格式

1、数据格式处理，也就是把我们的图片.jpg,.png等图片以及标注标签,打包在一起，搞成caffe可以直接方便调用的文件

2、编写网络结构文件，这个文件的后缀格式是.prototxt。就是编写你的网络有多少层，每一层有多少个特征图，输入、输出

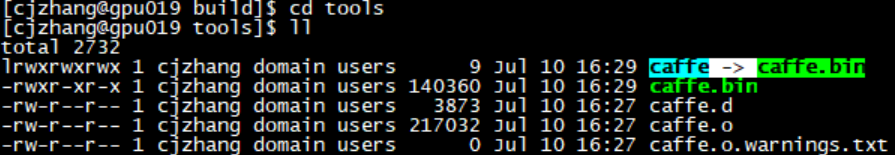
3、网络求解文件，这个文件我们喜欢把它取名为：solver.prototxt，这个文件的后缀格式也是.prototxt。这个文件主要包含梯度下降参数、迭代次数等参数……

4、编写网络求解文件后，我们可以说已经完成了CNN网络的编写。接着我们需要把这个文件，作为caffe的输入参数，调用caffe可执行文件，进行训练就可以了

首先caffe可执行文件，调用了solver.prototxt文件，而这个文件又调用了网络结构文件lenet\_train\_test.prototxt，然后lenet\_train\_test.prototxt文件里面又会调用输入的训练图片数据等。因此我们如果要训练自己的模型，需要备好3个文件：数据文件lmdb(该文件包含训练数据)、网络结构lenet\_train\_test.prototxt、求解文件solver.prototxt，这几个文件名随便，但是文件后缀格式不要随便乱改

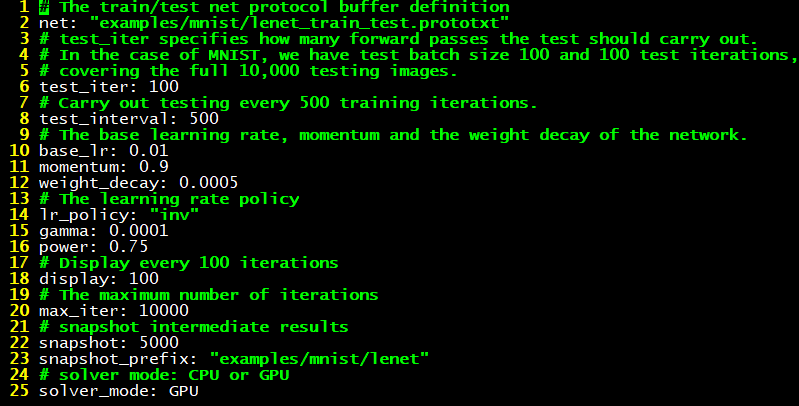
make -j //编译

make clean //清除



箭头是什么意思？

**网络求解文件solver.prototxt的参数含义：**



* net：

网络结构文件的路径

为什么要这样？

也可用train\_net和test\_net来对训练模型和测试模型分别设定。例如：

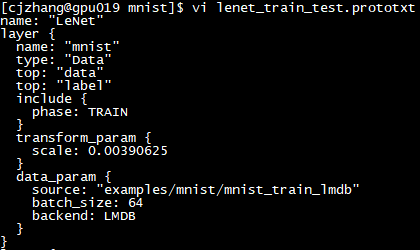
train\_net: "examples/hdf5\_classification/logreg\_auto\_train.prototxt"

test\_net: "examples/hdf5\_classification/logreg\_auto\_test.prototxt"

* test\_iter：

一次测试包含的迭代次数。网络一次迭代将一个batchSize的图片进行测试。为了能将test集中所有图片都测试一次，这个参数乘以test集的batchSize应该等测试集中图片总数量。测试集的batchsize在train\_test.prototxt文件中定义

在合理范围内，增大 Batch\_Size 有何好处？内存利用率提高了，大矩阵乘法的并行化效率提高



* test\_interval：

表示网络训练过程迭代多少次进行一次测试（计算一次测试误差）。caffe在训练的过程是边训练边测试的。比如这里设置的是224，即网络每迭代224次即对网络的准确率进行一次验证。一般来说，我们需要将训练集中所有图片都跑一遍，再对网络的准确率进行测试，因此整个参数乘以网络data层（TRAIN）中batchSize参数应该等于训练集中图片总数量。即test\_interval\*batchSize=train\_num。计算一次测试误差就需要包含所有的测试图片，使用test\_iter参数设置一次测试过程包含的迭代数

* base\_lr：

表示网络的基础学习率。学习率过高可能导致loss无法收敛等等问题。过低的学习率会使网络收敛慢。一般我们设置为0.01

* 学习率的调整策略：

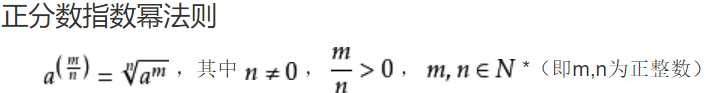
lr\_policy: "inv"

gamma: 0.0001

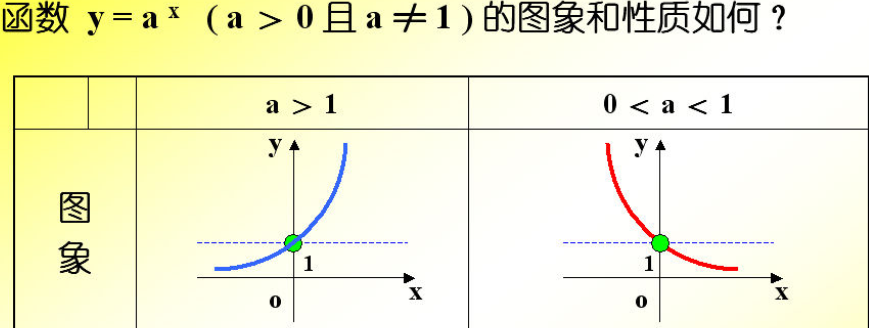
power: 0.75

当lr\_policy设置为inv（倒数衰减）时，还需要设置参数gamma和power（“次方“）,

返回base\_lr \* (1 + gamma \* iter) ^ (- power)，学习率随着迭代数增加而下降accepte



y=x的a次方，a>0，x从0开始增加，则y单调增加



* 正则化：

神经网络训练过程中逐渐overfitting时，网络权值逐渐变大

caffe提供了两种正则化，L1和L2，其中L2正则化项是默认存在的，在caffe.proto中可以找到。lenet\_solver.prototxt中可以自己设置为L1，添加一句代码regularization\_type: "L1"

* weight\_decay：

权重衰减项，防止过拟合。weight\_decay越大，则损失函数中正则项占比越大

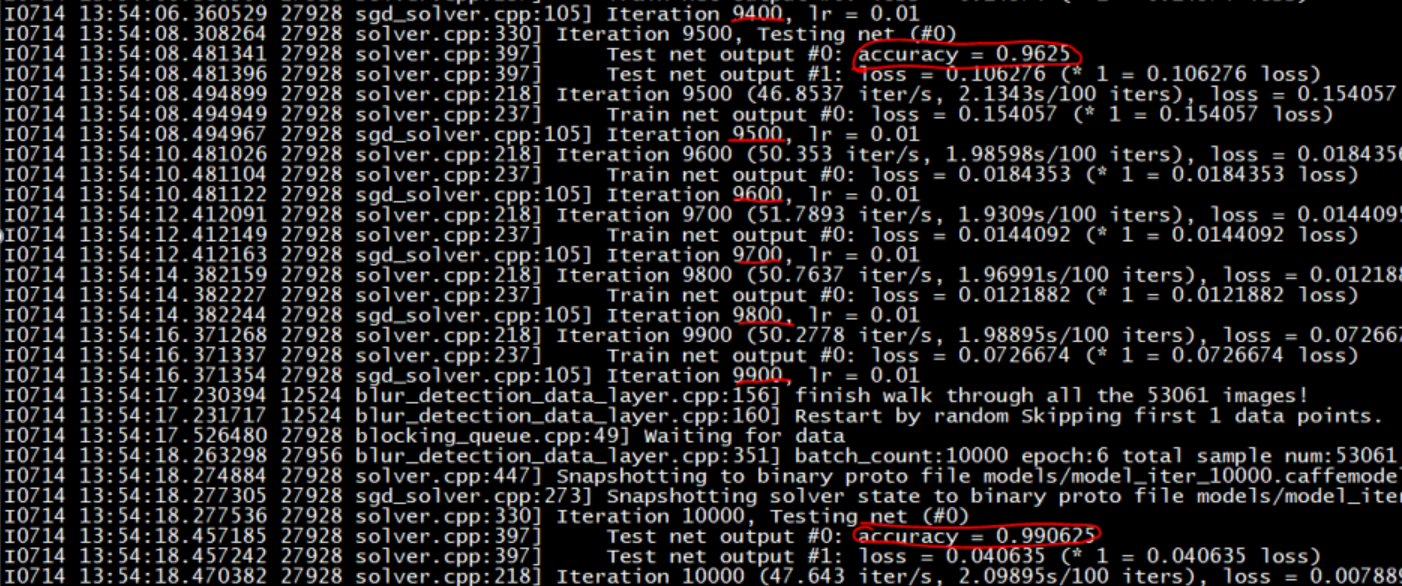
* 优化算法：

caffe提供六种优化算法，默认为SGD，可以自己设置为其它的，如type: AdaGrad

* display：

每隔多少代，在屏幕显示一次

当display设为100代，test\_interval设为500代时，效果如下图：



* max\_iter：

最大迭代次数。达到最大迭代次数时，训练结束

* snapshot：

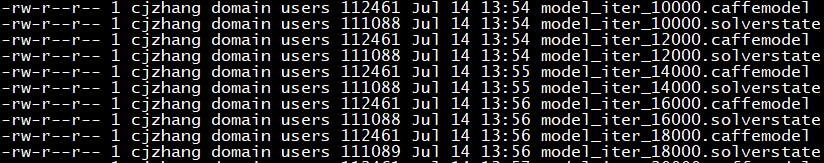
快照。将训练出来的model和solver状态进行保存，snapshot用于设置每训练多少代保存一次，默认为0，不保存。

* snapshot\_prefix：

设置快照的保存路径，最后一个/后的字符串是模型名的前缀，如：

设置snapshot\_prefix: "models/model"，已创建了models文件夹，则保存的模型在models文件夹下，名为model\_iter\_\*

snapshot设置为2000时，保存的模型和solver状态如下：



设置snapshot\_prefix: "/home/cjzhang/mnistExample/trainedmodels/"，已存在trainedmodels文件夹，则保存的模型在trainedmodels文件夹下，名为\_iter\_\*

设置snapshot\_prefix: "/home/cjzhang/mnistExample/trainedmodels"，已存在trainedmodels文件夹，则保存的模型在mnistExample文件夹下，名为trainedmodels \_iter\_\*

* solver\_mode:

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

// iter\_size也是在solver.prototxt里设置，默认为1。实际上的batch\_size=iter\_size\*网络定义里的batch\_size，因此每一次迭代的loss是iter\_size次迭代的和，再除以iter\_size，这个loss是通过调用`Net::ForwardBackward`函数得到的。这个设置我的理解是在GPU的显存不够的时候使用，比如我本来想把batch\_size设置为128，但是会out\_of\_memory，借助这个方法，可以设置batch\_size=32，iter\_size=4，那实际上每次迭代还是处理了128个数据，避免梯度爆炸，如果梯度的二范数超过了某个数值则进行scale操作，将梯度减小

加载模型：

int main(int argc, char\*argv[])

{

string model\_define = argv[3];

string model\_weight = argv[4];

FaceBlur blur\_model(model\_define,model\_weight);

}

运行时的命令：

./caffe/build/tools/blur\_demo.bin

blur\_data\_ldmk.txt

blur\_prediction.txt

blur\_train/netConfig/net\_deploy2.prototxt

blur\_train/models/model\_iter\_100000.caffemodel

result\_img/

void detect(Net<float>& net, Mat& img, int& label,float& conf)

{

setImg(net, img); #feed image to net，在哪定义的，什么作用

net.Forward(); # 前向传播，预测一次

auto outblob = net.blob\_by\_name("blur\_prob")->cpu\_data();

label= outblob[0]>outblob[1] ? 0: 1;

conf = outblob[1];

}

**protobuf:**

protoc -I=C:\Users\myth\Desktop --cpp\_out=C:\Users\myth\E\garbage addressbook.proto

如果没有将protoc添加到path环境变量，则需要进入protoc.exe文件所在目录再输入上边语句。若将protoc添加到path环境变量，则在任意位置都可以直接protoc

第一个参数是.proto文件的目录，第二个参数是生成.pb.h和.pb.cc文件的存储目录，第三个参数是.proto文件路径

getter函数具有与字段名一模一样的名字，并且是小写的，而setter函数都是以set\_前缀开头

clear\_前缀的函数，用来将字段重置（un-set）到空状态（empty state）

has\_前缀的函数，对每一个单一的（required或optional的）字段来说，如果字段被置（set）了值，该函数会返回true

// required int32 age = 2;

inline bool Person::has\_age() const {

return (\_has\_bits\_[0] & 0x00000004u) != 0;

}

inline void Person::set\_has\_age() {

\_has\_bits\_[0] |= 0x00000004u;

}

inline void Person::clear\_has\_age() {

\_has\_bits\_[0] &= ~0x00000004u;

}

inline void Person::clear\_age() {

age\_ = 0;

clear\_has\_age();

}

inline ::google::protobuf::int32 Person::age() const {

return age\_;

}

inline void Person::set\_age(::google::protobuf::int32 value) {

set\_has\_age();

age\_ = value;

}

重复的字段也有一些特殊的函数——如果你看一下重复字段phone 的那些函数，就会发现你可以：   
（1）得到重复字段的\_size（换句话说，这个Person关联了多少个电话号码）。

（2）通过索引（index）来获取一个指定的电话号码。

（3）通过指定的索引（index）来更新一个已经存在的电话号码。

（4）向消息（message）中添加另一个电话号码，然后你可以编辑它（重复的标量类型有一个add\_前缀的函数，允许你传新值进去）

// repeated int32 phone = 4;

int phone\_size() const;

void clear\_phone();

static const int kPhoneFieldNumber = 4;

::google::protobuf::int32 phone(int index) const;

void set\_phone(int index, ::google::protobuf::int32 value);

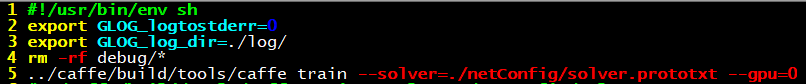
void add\_phone(::google::protobuf::int32 value);

const ::google::protobuf::RepeatedField< ::google::protobuf::int32 >&

phone() const;

只有add\_phone添加一个字段，.phone\_size()才会增加1

caffe的Command Line Interfaces一共提供了四个功能：train/test/time/device\_query，而Interfaces的输入除了这四种功能还可以输入诸如-solver/-weights/-snapshot/-gpu等参数



BrewFunction是一个函数指针的typedef，BrewMap是一个map<,>的typedef

typedef int (\*BrewFunction)();

typedef std::map<caffe::string, BrewFunction> BrewMap;

CHECK\_GT(FLAGS\_solver.size(), 0) << "Need a solver definition to train.";

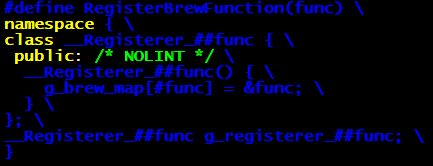
使用了glog的CHECK\_GT宏（含义为check greater than），检查FLAGS\_solver的size是否大于0



第二行代码是确保用户没有同时提供snapshot和weights参数，这两个参数都是继续之前的训练或者进行fine-tuning的，如果同时指明了这两个标志，则不知道到底应该从哪个路径的文件去读入模型的相关参数更为合适

caffe/tools/caffe.cpp中

实现了int train()、int test()、int time()、int device\_query()等四个函数，在每个函数实现后，借用宏定义



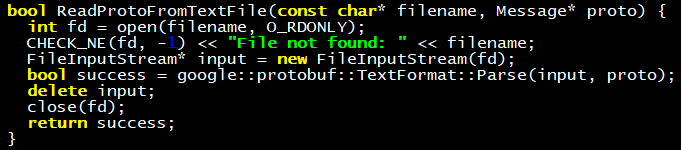
将字符串train、test、time、device\_query及对应的函数指针添加到全局变量BrewMap g\_brew\_map中，main()函数最终return参数对应的函数指针

其中，int train()函数

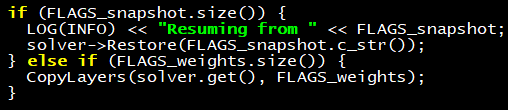
1、从网络求解文件.prototxt读取参数存入SolverParameter对象

caffe::SolverParameter solver\_param;

caffe::ReadSolverParamsFromTextFileOrDie(FLAGS\_solver, &solver\_param);



2、如果有snapshot和weights参数，则SolverParameter对象去读取已经训练好的网络的参数



**网络结构文件：**

数据层：

**type**: 层类型，如果是Data，表示数据来源于LevelDB或LMDB

**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。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

caffe的mirror参数是设置来对数据进行随机水平翻转，当设置为true，对于训练集中的每幅图片，都会有一定概率翻转，因为是随机翻转，所以同一副图可能在第二个epoch和第一个epoch分别是被翻转、未翻转，所以理想情况是相当于数据集double了

如果定义了crop\_size，那么在train时会对大于crop\_size的图片进行随机裁剪，而在test时只是截取中间部分

data\_param中backend: 选择是采用LevelDB还是LMDB, 默认是LevelDB

**数据来自于数据库（如LevelDB和LMDB）**：层类型（layer type）:Data

**数据来自于内存**：层类型：MemoryData

必须设置的参数：

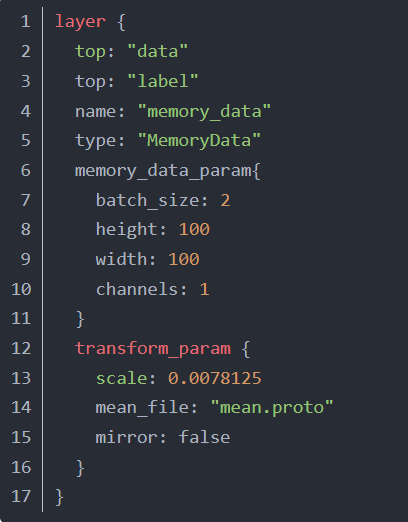
batch\_size：每一次处理的数据个数

channels：通道数

height：高度

width: 宽度

示例：



**数据来自于图片**：层类型：ImageData，imag：e\_data\_param中必须设置的参数：

  source: 一个文本文件的名字，每一行给定一个图片文件的名称和标签（label)

  batch\_size: 每一次处理的数据个数，即图片数

image\_data\_param中optional bool is\_color = 11 [default = true];

optional bool shuffle\_images = 64 [default = false];

shuffle开了叫 随机梯度下降，不开叫梯度下降。从优化角度来说区别巨大。实际角度来说，不shuffle的话神经网络会记住特定的example顺序

可选参数：

  rand\_skip: 在开始的时候，路过某个数据的输入。通常对异步的SGD很有用。

  shuffle: 随机打乱顺序，默认值为false

  new\_height,new\_width: 如果设置，则将图片进行resize

示例：

1. layer {
2. name: "data"
3. type: "ImageData"
4. top: "data"
5. top: "label"
6. transform\_param {
7. mirror: **false**
8. crop\_size: 227
9. mean\_file: "data/ilsvrc12/imagenet\_mean.binaryproto"
10. }
11. image\_data\_param {
12. source: "examples/\_temp/file\_list.txt"
13. batch\_size: 50
14. new\_height: 256
15. new\_width: 256
16. }
17. }

卷积层Convolution

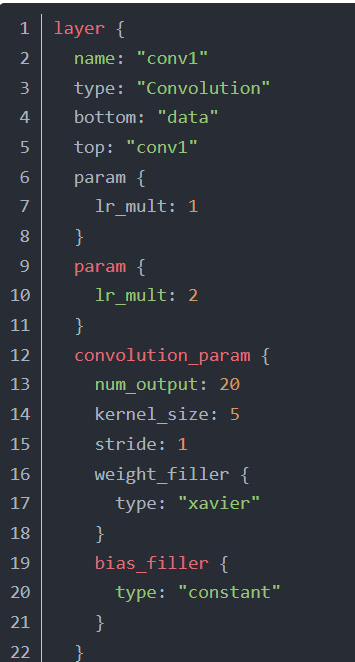
视觉层（Vision Layers)包括Convolution, Pooling, Local Response Normalization (LRN), im2col等层

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

num\_output: 卷积核（filter)的个数

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



在caffe中，卷积运算就是先对数据进行im2col操作，再进行内积运算（inner product)。这样做，比原始的卷积操作速度更快

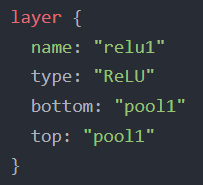
激活层：

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

一般来说都是先非线性的激活函数，然后再加pooling层。但是这里如果是Relu和max pooling的话，交换次序无影响，因为：

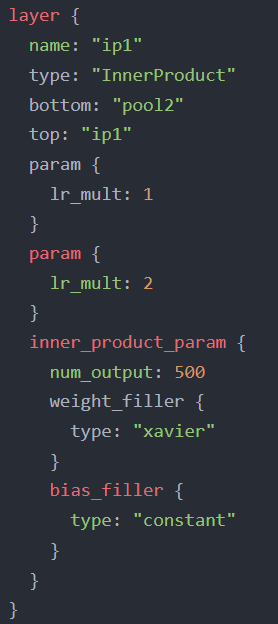


有的网络卷积层后不一定有激活函数，如caffe的LeNet，卷积—池化--卷积—池化—全连接—ReLU—全连接—SoftmaxWithLoss：



全连接层Inner Product：

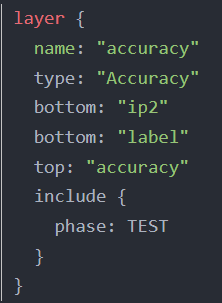
Inner Product全连接层实际上也是一种卷积层，只是它的卷积核大小和原数据大小一致。因此它的参数基本和卷积层的参数一样



**Accuracy：**

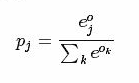
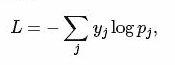
输出分类（预测）精确度，只有test阶段才有，因此需要加入include参数

caffe中计算Accuracy时，是通过比较最后一个全连接层（神经元个数=类别数、但没有加入activation function）的输出和数据集的labels来得到的。在Accuracy层内部，实现了“利用最后一个全连接层的输出得到数据集的预测labels”（数值最大的那个值得idnex就是样本的类别），那么，再与输入的数据集的真实lebels作对比，就实现了Accuracy的计算

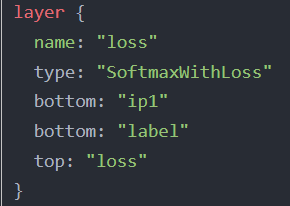
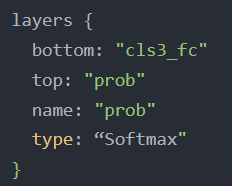


**Softmax或SoftmaxWithLoss：**

Softmax计算公式： SoftmaxWithLoss的计算公式：

不管是softmax layer还是softmax-loss layer,都是没有参数的

* 实际上，如果仅仅是做预测，利用最后一个全连接层的输出就够了（输出值最大的那个位置即为输入的label），该输出表示了输入的样本属于每一类的可能性大小，但并不是概率值；
* 如果为了使输出具有统计意义，需要加入Softmax层，它只是使前面的全连接层的输出（最后一个全连接层）具有了概率意义，并不改变这些输出之前的大小关系，因为softmax function本身就是增函数；
* 为了利用误差反向传播，还需要构造loss function，需要利用softmax function的输出，即需要利用输入样本属于每一类的概率值，即SoftmaxWithLoss层

一般分类问题训练时最后一层为SoftmaxWithLoss层，使用训练好的模型时最后一层为Softmax层

Slice层：

假设input的维度是N\*5\*H\*W, tops输出的维度分别是N\*1\*H\*W，N\*2\*H\*W、N\*1\*H\*W、N\*1\*H\*W 。 则prototxt文件：

layer {

name: "slice"

type: "Slice"

bottom: "input"

top: "output1"

top: "output2"

top: "output3"

top: "output4"

slice\_param {

axis: 1

slice\_point: 1

slice\_point: 3

slice\_point: 4

}

slice\_point 的含义1、2、3、4、5，slice\_point是1、3、4，因此是1和2+3和4和5

这里需要注意的是，如果有slice\_point，slice\_point的个数一定要等于top的个数减一。   
axis表示要进行分解的维度。

slice\_point的作用是将axis按照slic\_point 进行分解。   
slice\_point没有设置的时候则对axis进行均匀分解

**生成LMDB格式的数据：**

1. [cjzhang@gpu019 LMDBExample]$ vi create\_imagenet.sh
2. set -e  #若指令传回值不等于0，则立即退出shell
4. EXAMPLE=/home/cjzhang/LMDBExample   #存放生成的LMDB数据文件夹
5. DATA=/home/cjzhang/LMDBExample  #生成的文件列表.txt文件所在文件夹
6. TOOLS=/data/zhangchangjian/caffe/build/tools #caffe的工具库
7. #TRAIN\_DATA\_ROOT=
8. VAL\_DATA\_ROOT=/home/cjzhang/LMDBExample/rbnames/
10. # Set RESIZE=true to resize the images to 256x256. Leave as false if images have
11. # already been resized using another tool.
12. RESIZE=false
13. **if** $RESIZE; then
14. RESIZE\_HEIGHT=256
15. RESIZE\_WIDTH=256
16. **else**
17. RESIZE\_HEIGHT=0
18. RESIZE\_WIDTH=0
19. fi
21. #if [ ! -d "$TRAIN\_DATA\_ROOT" ]; then
22. # echo "Error: TRAIN\_DATA\_ROOT is not a path to a directory: $TRAIN\_DATA\_ROOT"
23. # echo "Set the TRAIN\_DATA\_ROOT variable in create\_imagenet.sh to the path" \
24. #    "where the ImageNet training data is stored."
25. # exit 1
26. #fi
28. **if** [ ! -d "$VAL\_DATA\_ROOT" ]; then
29. echo "Error: VAL\_DATA\_ROOT is not a path to a directory: $VAL\_DATA\_ROOT"
30. echo "Set the VAL\_DATA\_ROOT variable in create\_imagenet.sh to the path" \
31. "where the ImageNet validation data is stored."
32. exit 1
33. fi
35. #echo "Creating train lmdb..."
37. #GLOG\_logtostderr=1 $TOOLS/convert\_imageset \
38. #    --resize\_height=$RESIZE\_HEIGHT \
39. #    --resize\_width=$RESIZE\_WIDTH \
40. #    --shuffle \
41. #    $TRAIN\_DATA\_ROOT \
42. #    $DATA/train.txt \
43. #    $EXAMPLE/ilsvrc12\_train\_lmdb
45. echo "Creating val lmdb..."
47. rm -rf $EXAMPLE/test\_lmdb                 #删除$EXAMPLE/test\_lmdb文件夹，-f即要删除的对象不存在时，执行该命令不报错
49. GLOG\_logtostderr=1 $TOOLS/convert\_imageset \
50. --resize\_height=$RESIZE\_HEIGHT \
51. --resize\_width=$RESIZE\_WIDTH \
52. --shuffle \
53. $VAL\_DATA\_ROOT \
54. $DATA/test.txt \
55. $EXAMPLE/test\_lmdb
57. echo "Done."
59. #图片完整路径被认为是：$VAL\_DATA\_ROOT+test.txt中某一行，构成的一个字符串
60. #变量DATA和文件名，共同组成.txt文件的完整路径
61. #在变量EXAMPLE所指路径下，创建test\_lmdb文件夹，存放生成的LMDB数据

**GPU ID 号使用：**

./build/tools/caffe train --solver=examples/testXXX/solver.prototxt

当电脑中有多个GPU时，默认使用GPU0。如果想使用其他的GPU，可以将该文件内容修改如下：

./build/tools/caffe train --solver=examples/testXXX/solver.prototxt --gpu 2

注意，caffe中默认编号从0开始，因而--GPU 2的意思是使用第3个GPU

如果要使用多个GPU，可以使用如下命令：

./build/tools/caffe train --solver=examples/testXXX/solver.prototxt --gpu 0,1,2,3

如果要使用所有的GPU，可使用如下命令：

./build/tools/caffe train --solver=examples/testXXX/solver.prototxt --gpu all

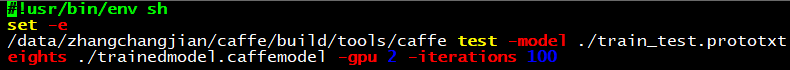
**caffe的tools/和build/tools/**

caffe的C++主程序（caffe.cpp)放在根目录下的tools文件夹内, 当然还有一些其它的功能文件，如：convert\_imageset.cpp, upgrade\_net\_proto\_text.cpp等也放在这个文件夹内。

经过编译后，这些文件都被编译成了可执行文件，放在了 ./build/tools/ 文件夹内，如caffe.bin，convert\_imageset.bin，upgrade\_net\_proto\_binary.bin。因此我们要执行caffe程序，都需要加 ./build/tools/ 前缀

**caffe test用法：**

[cjzhang@gpu019 OnlyTest]$ vi testFromLMDB.sh



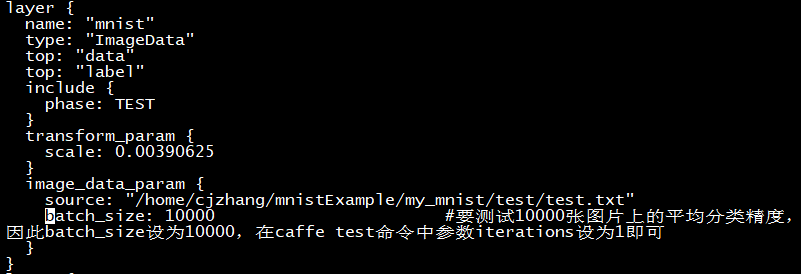
这个例子比较长，不仅用到了test参数，还用到了-model, -weights, -gpu和-iteration四个参数。意思是利用训练好了的权重（-weight)，输入到测试模型中(-model)，用编号为2的gpu(-gpu)测试1次(-iteration)

将要测试的样本组织为合适的类型，如Data或ImageData，然后修改网络结构文件中用于测试阶段的数据层

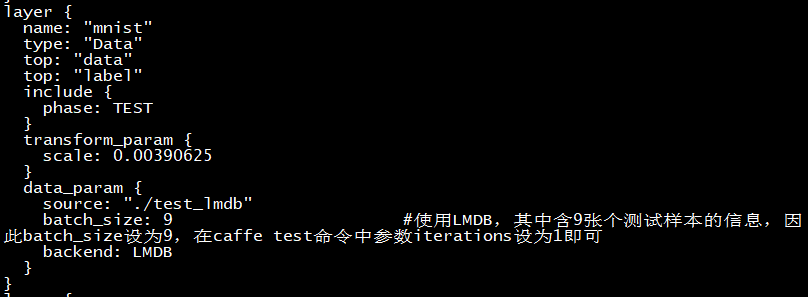
将batch\_size设置为要测试的数据量

test命令中迭代数参数设置为1

这样，测试时一个batch，恰好是整个测试数据集的大小



或者



**caffe中的宏CHECK\_\***

#define CHECK\_EQ(val1, val2) CHECK\_OP(\_EQ, ==,val1, val2) //相当于assert(val1 == val2) release下可用,如果检测为true，则返回NULL，否则就会返回一个有明确提示信息的字符串指针，并输出该信息，然后是程序宕掉。以下同理

#define CHECK\_NE(val1, val2) CHECK\_OP(\_NE, !=,val1, val2) //相当于assert(val1 != val2)

#define CHECK\_LE(val1, val2) CHECK\_OP(\_LE, <=,val1, val2) //相当于assert(val1 <= val2)

#define CHECK\_LT(val1, val2) CHECK\_OP(\_LT, < ,val1, val2) // 相当于assert(val1 < val2)

#define CHECK\_GE(val1, val2) CHECK\_OP(\_GE, >=,val1, val2) //相当于assert(val1 >= val2)

#define CHECK\_GT(val1, val2) CHECK\_OP(\_GT, > ,val1, val2) //相当于assert(val1 > val2)

**使用训练好的caffe模型预测：?**

const string& model\_file, //网络结构.prototxt文件路径

const string& trained\_file, //训练好的caffemodel文件路径

share\_ptr<Net<float>> net\_；

net\_.reset(new Net<float>(model\_file,TEST)); //从model\_file路径下的prototxt初始化网络结构

net->CopyTrainedLayersFrom(trained\_file); //读入训练完毕的网络参数

net\_->num\_inputs()

net\_->num\_outputs()

Blob<float>\* input\_layer = net\_->input\_blobs()[0]; //获取网络输入的blob，表示网络的数据层

input\_layer->channels() //网络要求的输入通道数

input\_layer->width() //网络要求的输入图像宽

input\_layer->height() //网络要求的输入图像高

Blob<float>\* output\_layer = net\_->output\_blobs()[0]; //输出层

const float\* begin = output\_layer->cpu\_data();

const float\* end = begin + output\_layer->channels();

return std::vector<float>(begin, end);

Caffe::set\_mode(Caffe::GPU)

Caffe::set\_mode(Caffe::CPU)

局部排序：

static bool PairCompare(const std::pair<float, int>& lhs, const std::pair<float, int>& rhs) {

return lhs.first > rhs.first;

}

#include<utility>

借助pair实现获取排序后元素的索引

/\* Return the indices of the top N values of vector v. \*/

static std::vector<int> Argmax(const std::vector<float>& v, int N) {

std::vector<std::pair<float, int> > pairs;

for (size\_t i = 0; i < v.size(); ++i)

pairs.push\_back(std::make\_pair(v[i], i));

std::partial\_sort(pairs.begin(), pairs.begin() + N, pairs.end(), PairCompare); //局部排序，使元素从前到后满足PairCompare，仅使[pairs.begin()，pairs.begin()+N）有序

std::vector<int> result;

for (int i = 0; i < N; ++i)

result.push\_back(pairs[i].second);

return result;

}

Blob<float>\* input\_layer = net\_->input\_blobs()[0];

input\_layer->Reshape(1, num\_channels\_, input\_geometry\_.height, input\_geometry\_.width);

/\* Forward dimension change to all layers. \*/

net\_->Reshape();

std::vector<cv::Mat> input\_channels;

WrapInputLayer(&input\_channels);//将net\_->input\_blobs()[0]->mutable\_cpu\_data()分离为多个通道的数据，input\_channels[0].data等于net\_->input\_blobs()[0]->mutable\_cpu\_data()

Preprocess(img, &input\_channels); //对输入图像的高、通道数、数据类型调整，减去mean\_，写入input\_channels所指地址

net\_->Forward();

**数据预处理：**

特征归一化的3种方法：

* 简单缩放，如mnist数据集
* 逐样本均值削减(也称为移除直流分量)
* 特征标准化(减均值，除以标准差)

减均值除以方差的方式，在Batch Normalization(BN)出现之前是很有必要的，学习的时候更加容易。但是BN出现之后，这个操作就完全没必要了。

mnist的预处理：

mnist只简单缩放，归一化到0到1的范围

如果梯度非常大，学习率就必须非常小（否则会跳过local minimum），因此，学习率（学习率初始值）的选择需要参考输入层的数值，不如直接将数据归一化，这样学习率就不必再根据数据范围作调整

减去图像均值和像素均值：

* image mean：读入一张彩色图像，假设是(N\*N\*3)，这时候，求出image mean的话，就也是N\*N\*3，相当于把所有训练集在同一个空间位置上的像素的对应通道求了均值。
* pixel mean：其实是把训练集里面所有图片的所有R通道像素，求了均值，G,B通道类似，也就是不考虑空间位置了。所以求出来就是三个数值（R\_mean,G\_mean,B\_mean）

caffe的compute\_image\_mean.cpp文件求image mean保存到mean.binaryproto文件（同时在程序的最后又根据image mean求出pixel mean并LOG(INFO)出来）

但使用训练好的模型预测时，给的classification.cpp文件中对输入图片减去的均值是pixel mean，而不是image mean，减去的pixel mean由mean.binaryproto文件计算得到。减去pixel mean后送入网络输入层net\_->input\_blobs()[0]

1. //classification.cpp
2. Blob<**float**> mean\_blob;
3. ……//从mean.binaryproto文件读取数据赋值mean\_blob，mean.binaryproto文件数据类型是32-bit float BGR or grayscale
4. std::vector<cv::Mat> channels;
5. **float**\* data = mean\_blob.mutable\_cpu\_data();
6. **for** (**int** i = 0; i < mean\_blob.channels(); i++)
7. {
8. cv::Mat temp(mean\_blob.height(), mean\_blob.width(), CV\_32FC1, data);
9. channels.push\_back(temp);
10. data += mean\_blob.height()\*mean\_blob.width();
11. }
13. cv::Mat fuse;
14. cv::merge(channels, fuse);  //多个通道的图像融合为一幅
15. cv::Scalar channel\_mean = cv::mean(fuse); //各个通道的均值
16. cv::Mat mean\_(fuse.size(), fuse.type(), channel\_mean);
17. ……//对要预测的输入图像通道调整、resize、转为float得到sample\_normalized
18. cv::Mat sample\_normalized;
19. cv::subtract(sample\_float, mean\_, sample\_normalized);

自然图像与非自然图像：

自然图像一般都会减去均值

神经网络要求：(1)特征的均值大致为0；(2)不同特征的方差值彼此相似。但同时要求输入不能太大，因此一还需要将输入处理到[0,1]范围。如

1. 对于自然图像，只需要图像零均值化，不需要做特征标准化。因为自然图片具有平稳性，图像任一部分的统计性质和其它部分相同，即不同位置的像素点的同一通道有相同的均值和方差。所以对自然图像训练集，减去图像均值约等于减去像素均值。所以对自然图像训练集，不需要进行任何方差归一化操作，唯一还需进行的规整化操作就是均值规整化。减去像素均值即可。实际上是移除图像的平均亮度值 (intensity)。根据应用，在大多数情况下，我们并不关注所输入图像的整体明亮程度。比如在对象识别任务中，图像的整体明亮程度并不会影响图像中存在的是什么物体，所以可以减去平均亮度值是有意义的。

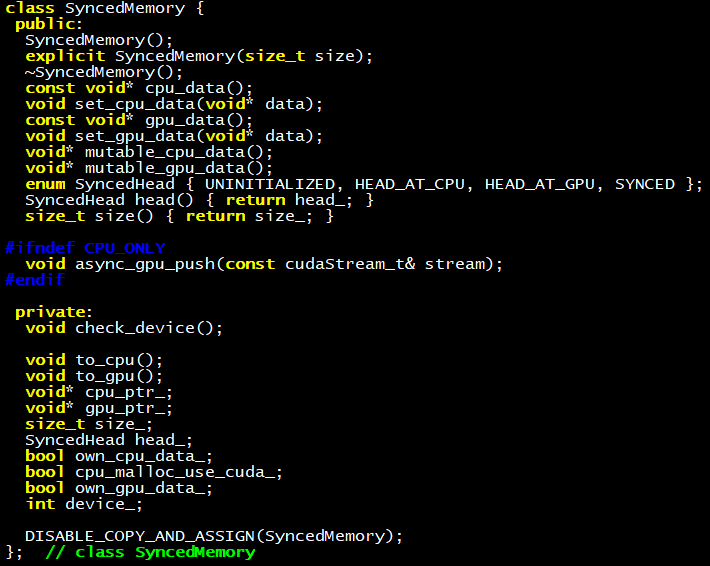
（2）如果处理的图像并非自然图像（比如，手写文字，或者白背景正中摆放单独物体），其他归一化操作就值得考虑了

# caffe Blob：

class SyncedMemory 的bool cpu\_malloc\_use\_cuda\_数据成员是什么意思？

enum SyncedHead { UNINITIALIZED, HEAD\_AT\_CPU, HEAD\_AT\_GPU, SYNCED };

head\_指示最新的数据是在内存中，还是在显存中，还是内存和显存处于同步状态



#ifndef CPU\_ONLY

NO\_GPU;

to\_cpu()：

* 如果head\_是UNINITIALIZED，则分配内存空间，memset为0
* 如果head\_是HEAD\_AT\_GPU，则分配内存空间（如果cpu\_ptr\_不为NULL则不需要分配），cpu\_ptr\_指向之，将显存数据拷贝到内存，head\_设置为SYNCED
* 如果head\_是HEAD\_AT\_CPU或SYNCED，则不用处理

to\_gpu()：

* 如果head\_是UNINITIALIZED，则分配显存空间，memset为0
* 如果head\_是HEAD\_AT\_CPU，则分配显存空间（如果cpu\_ptr\_不为NULL则不需要分配），gpu\_ptr\_指向之，将内存数据拷贝到显存，head\_设置为SYNCED
* 如果head\_是HEAD\_AT\_GPU或SYNCED，则不用处理

const void\* SyncedMemory::cpu\_data() {

check\_device();

to\_cpu();

return (const void\*)cpu\_ptr\_;

}

void\* SyncedMemory::mutable\_cpu\_data() {

check\_device();

to\_cpu();

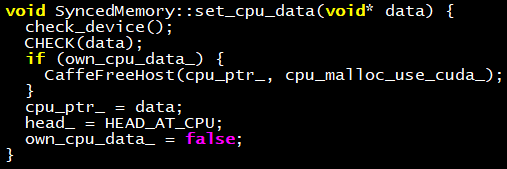
head\_ = HEAD\_AT\_CPU;

return cpu\_ptr\_;

}

调用mutable\_cpu\_data()可以让head转移到cpu上

经验上来说，如果不需要改变其值，则使用常量调用的方式。因为这样设计能减少数据传递从而提高效率。SyncedMem会决定何时去复制数据，通常情况是仅当gpu或cpu修改后有复制操作。比如，之前head\_是SYNCED，表示cpu\_ptr\_和gpu\_ptr\_所指的内存和显存中数据一致，调用mutable\_cpu\_data()，由于可能修改内存数据（实际可能并没有修改），因此head\_被置为HEAD\_AT\_CPU；然后下次调用gpu\_data()或mutable\_gpu\_data()时，需要从内存拷贝数据到显存。当调用cpu\_data()时，则显示告诉不修改内存数据，不修改head\_。



Blob本质是对SyncedMemory的再封装A wrapper around SyncedMemory

Blob同时保存了data\_和diff\_，其类型为SyncedMemory的指针

Blob就是一个N维数组，N可以是2，3，4等

C-contiguous方式来存储，行占优（row-major）方式存储的

因此，对于(n, k, h, w)的位置，其物理内存地址即为：((n×K+k)×H+h)×W+w

在模型中设定的参数，也是用Blob来表示和运算。比如：在一个卷积层中，输入一张3通道图片，有96个卷积核，每个核大小为11\*11，因此这个Blob是96\*3\*11\*11. 而在一个全连接层中，假设输入1024通道图片，输出1000个数据，则Blob为1000\*1024

分类网络中分类器类别有10类，那么，输出的blob就会有10个通道，每个通道的长宽都为1(因为是10个数

Blob可以在CPU与GPU之间同步数据，提供了统一的存储接口来处理数据

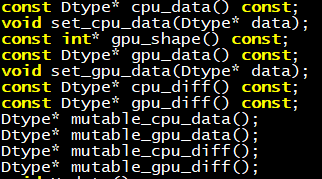
Blob存储的数据分为两类：data，diff，前者指的是我们在网络间正常交流的数据，如图像像素数据之类的；后者指的是Back propagation进行学习时，运算得到的梯度数据

由于这些数据可以在CPU上计算、也可以在GPU上运算，因此，要想获取这些数据分别有两种方式。拿在CPU上运算的data数据举例：

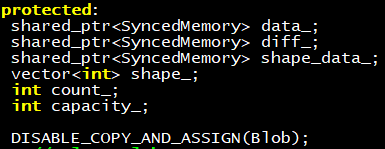
const Dtype\* cpu\_data() const; //不可改变数据值

Dtype\* mutable\_cpu\_data(); //可以改变数据值

如果不需要改变数据的值，尽可能使用const访问的方式



数据成员：



shape\_为blob维度

类Blob有数据成员shared\_ptr<SyncedMemory> data\_，构造函数中使用初始化列表初始化data\_()，什么语法？data\_()是类shared\_ptr<>的对象，data\_()表示调用类的无参构造函数

void Blob<Dtype>::Reshape(const vector<int>& shape)函数：

shape\_.resize(shape.size());

if (!shape\_data\_ || shape\_data\_->size() < shape.size() \* sizeof(int)) {

shape\_data\_.reset(new SyncedMemory(shape.size() \* sizeof(int)));

}

int\* shape\_data = static\_cast<int\*>(shape\_data\_->mutable\_cpu\_data());

count\_ \*= shape[i]; //blob中的元素数量

shape\_[i] = shape[i];

shape\_data[i] = shape[i];

if (count\_ > capacity\_) {

capacity\_ = count\_;

data\_.reset(new SyncedMemory(capacity\_ \* sizeof(Dtype)));

diff\_.reset(new SyncedMemory(capacity\_ \* sizeof(Dtype)));

}

\* This function can be called both to create an initial allocation

\* of memory, and to adjust the dimensions of a top blob during Layer::Reshape

\* or Layer::Forward. When changing the size of blob, memory will only be

\* reallocated if sufficient memory does not already exist, and excess memory

\* will never be freed.

\*

\* Note that reshaping an input blob and immediately calling Net::Backward is

\* an error; either Net::Forward or Net::Reshape need to be called to

\* propagate the new input shape to higher layers.

Blob有三个SyncedMemory指针，每个SyncedMemory对象管理一块内存+一块显存，内存显存大小一致，数据可以同步

Blob中data，diff两个SyncedMemory指针管理的空间容量，由Blob的数据成员capacity\_记录，等于capacity\_\*sizeof(Dtype)。其中存储有效数据的空间大小由Blob的数据成员count\_记录，等于count\_\*sizeof(Dtype)。

Blob::reshape，更新count\_，如果新数据需要的空间大于capacity\_\*sizeof(Dtype)，则重新分配空间，SyncedMemory::SyncedMemory(size\_t size)，让data，diff两个SyncedMemory指针管理；如果新数据需要的空间小于capacity\_\*sizeof(Dtype)，则data，diff两个SyncedMemory指针管理的空间不变

data\_.size()永远等于capacity\_ \* sizeof(Dtype)

capacity\_永远>=count\_

template <typename Dtype>

void Blob<Dtype>::set\_cpu\_data(Dtype\* data) {

CHECK(data);

// Make sure CPU and GPU sizes remain equal

size\_t size = count\_ \* sizeof(Dtype);

if (data\_->size() != size) {

data\_.reset(new SyncedMemory(size));

diff\_.reset(new SyncedMemory(size));

}

data\_->set\_cpu\_data(data);

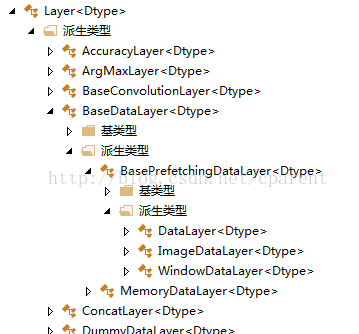
}

调用Blob<Dtype>::Reshape()后，data\_管理的空间大小为data\_.size()=capacity\_\*sizeof(Dtype)，而capacity>=count\_。再调用Blob<Dtype>:: set\_cpu\_data(Dtype\* data)，则一定要使data\_.size()=count\_\* sizeof(Dtype)。这时候为什么不更新capacity\_，capacity\_可能大于实际data\_管理的空间？

**caffe Layer：**

需按照src/caffe/proto/caffe.proto中定义的网络及参数格式定义网络 prototxt文件

每一个layer都是继承基类Layer的，其中BaseDataLayer是输入数据的基类



Debug技巧 ：

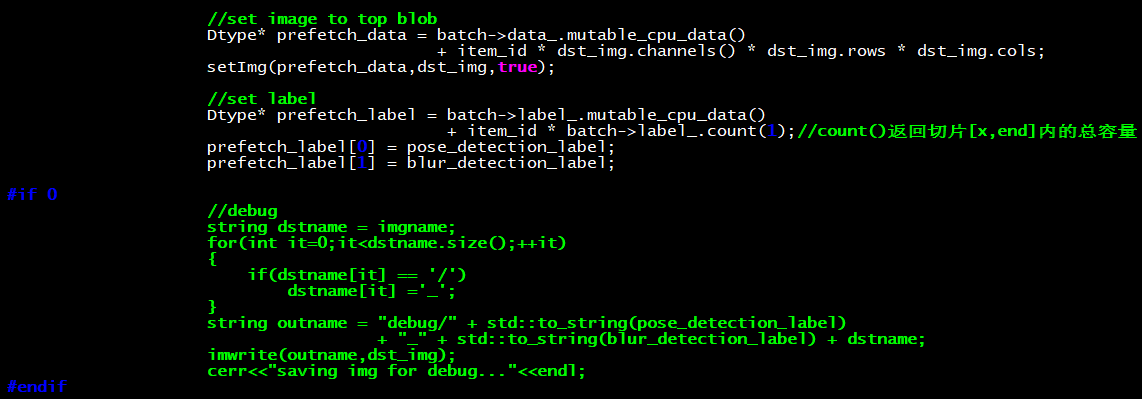
（1）在程序里

#if 1

……

#end if

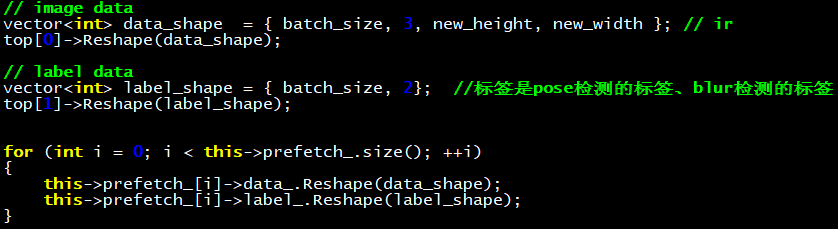
(2)将读入batch的img及标签保存：



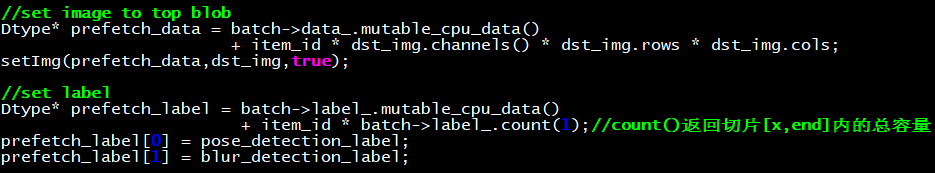
void HeadPoseBlurDataLayer<Dtype>::DataLayerSetUp(const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top)函数中：

(1)从prototxt文件获取参数new\_height、new\_width、batch\_size等

(2)设置top的shape：



void HeadPoseBlurDataLayer<Dtype>::load\_batch(Batch<Dtype>\* batch)函数将网络输入层要求的数据和标签赋值给网络Blob：



net->num\_inputs()和

net->num\_outputs()是什么含义