[caffe源码分析--math\_functions.cu代码研究](http://blog.csdn.net/lingerlanlan/article/details/24379465)

其中用到一个宏定义CUDA\_KERNEL\_LOOP

在common.hpp中有。

**#define**CUDA\_KERNEL\_LOOP(i,n) \

**for**(**int**i = blockIdx.x \* blockDim.x + threadIdx.x; \

i < (n); \

i +=blockDim.x \* gridDim.x)

先看看caffe采取的线程格和线程块的维数设计，

还是从common.hpp可以看到

CAFFE\_CUDA\_NUM\_THREADS

**CAFFE\_GET\_BLOCKS**(**constint**N)

明显都是一维的。

整理一下CUDA\_KERNEL\_LOOP格式看看，

**for**(**int**i = blockIdx.x \* blockDim.x + threadIdx.x;

i< (n);

i+= blockDim.x \* gridDim.x)

blockDim.x\* gridDim.x表示的是该线程格所有线程的数量。

n表示核函数总共要处理的元素个数。

有时候，n会大于blockDim.x\* gridDim.x，因此并不能一个线程处理一个元素。

由此通过上面的方法，让一个线程串行（for循环）处理几个元素。

这其实是常用的伎俩，得借鉴学习一下。

再来看一下这个核函数的实现。

template<typename Dtype>

\_\_global\_\_void mul\_kernel(const int n, const Dtype\* a,

constDtype\* b, Dtype\* y)

{

CUDA\_KERNEL\_LOOP(index,n)

{

y[index]= a[index] \* b[index];

}

}

明显就是算两个向量的点积了。

由于向量的维数可能大于该kernel函数线程格的总线程数量。

因此有些线程可以要串行处理几个元素。

[caffe源码分析--SyncedMemory类代码研究](http://blog.csdn.net/lingerlanlan/article/details/24379607)

数据成员：

**void**\*cpu\_ptr\_;数据在cpu的指针

**void**\*gpu\_ptr\_;数据在gpu的指针

size\_tsize\_;数据的大小

SyncedHeadhead\_;表示数据的状态，有四种状态，分别是未初始化，数据在cpu中，数据在gpu中，数据在cpu和gpu中都有

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

构造函数

**SyncedMemory**()

:cpu\_ptr\_(NULL),gpu\_ptr\_(NULL),size\_(0),head\_(*UNINITIALIZED*){}

简单的初始化

**explicitSyncedMemory**(size\_tsize)

:cpu\_ptr\_(NULL),gpu\_ptr\_(NULL),size\_(size),head\_(*UNINITIALIZED*){}

只是把size（大小）设置了，并未申请内存

析构函数

**SyncedMemory::~SyncedMemory**(){

//如果cpu有数据，则释放

**if**(cpu\_ptr\_){

CaffeFreeHost(cpu\_ptr\_);

}

//如果gpu有数据，则释放

**if**(gpu\_ptr\_){

CUDA\_CHECK(cudaFree(gpu\_ptr\_));

}

}

函数**voidto\_cpu**()

功能：把数据放到cpu上

1数据未初始化，则在cpu申请内存。此时状态为*HEAD\_AT\_CPU*

2数据本来在gpu，则从gpu拷贝内存到cpu。此时状态为*SYNCED*

*3数据本来在cpu，不做处理*

*4数据在cpu和gpu都有，不做处理*

**inlinevoidSyncedMemory::to\_cpu**(){

**switch**(head\_){

**case***UNINITIALIZED*:

CaffeMallocHost(&cpu\_ptr\_,size\_);

memset(cpu\_ptr\_,0,size\_);

head\_=*HEAD\_AT\_CPU*;

**break**;

**case***HEAD\_AT\_GPU*:

**if**(cpu\_ptr\_==NULL){

CaffeMallocHost(&cpu\_ptr\_,size\_);

}

CUDA\_CHECK(cudaMemcpy(cpu\_ptr\_,gpu\_ptr\_,size\_,cudaMemcpyDeviceToHost));

head\_=*SYNCED*;

**break**;

**case***HEAD\_AT\_CPU*:

**case***SYNCED*:

**break**;

}

}

函数**voidto\_gpu**();

功能：把数据放到gpu上

1数据未初始化，在gpu申请内存。此时状态为*HEAD\_AT\_GPU*

2数据在cpu，从cpu拷贝到gpu。此时状态为*SYNCED*

3数据在gpu，不做操作。

4数据在cpu和gpu都有，不做操作。

**inlinevoidSyncedMemory::to\_gpu**(){

**switch**(head\_){

**case***UNINITIALIZED*:

CUDA\_CHECK(cudaMalloc(&gpu\_ptr\_,size\_));

CUDA\_CHECK(cudaMemset(gpu\_ptr\_,0,size\_));

head\_=*HEAD\_AT\_GPU*;

**break**;

**case***HEAD\_AT\_CPU*:

**if**(gpu\_ptr\_==NULL){

CUDA\_CHECK(cudaMalloc(&gpu\_ptr\_,size\_));

}

CUDA\_CHECK(cudaMemcpy(gpu\_ptr\_,cpu\_ptr\_,size\_,cudaMemcpyHostToDevice));

head\_=*SYNCED*;

**break**;

**case***HEAD\_AT\_GPU*:

**case***SYNCED*:

**break**;

}

}

函数**constvoid**\***cpu\_data**();

功能：返回数据在cpu的指针

**constvoid**\***SyncedMemory::cpu\_data**(){

to\_cpu();

**return**(**constvoid**\*)cpu\_ptr\_;

}

函数**constvoid**\***gpu\_data**();

功能：返回数据在gpu的指针

**constvoid**\***SyncedMemory::gpu\_data**(){

to\_gpu();

**return**(**constvoid**\*)gpu\_ptr\_;

}

函数**void**\***mutable\_cpu\_data**();

功能：返回数据在cpu的指针，并改变数据的状态为*HEAD\_AT\_CPU*

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

to\_cpu();

head\_=*HEAD\_AT\_CPU*;

**return**cpu\_ptr\_;

}

函数**void**\***mutable\_gpu\_data**();

功能：返回数据在cpu的指针，并改变数据的状态为*HEAD\_AT\_GPU*

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

to\_gpu();

head\_=*HEAD\_AT\_GPU*;

**return**gpu\_ptr\_;

}

函数SyncedHead**head**(){**return**head\_;}

功能：返回数据的状态

函数size\_t**size**(){**return**size\_;}

功能：返回数据的大小

DISABLE\_COPY\_AND\_ASSIGN(**SyncedMemory**);

一个宏，把该类的拷贝函数和等号操作符给禁止掉

其实就是

**private**:\

SyncedMemory(**const**SyncedMemory&);\

SyncedMemory&**operator**=(**const**SyncedMemory&)

如果想让你的类不能使用copy构造函数和赋值操作符，只要将该类的copy构造函数和赋值操作符函数定义为private即可，并且只是声明，不用实现.

[caffe源码分析--Blob类代码研究](http://blog.csdn.net/lingerlanlan/article/details/24379689)

数据成员

shared\_ptr<SyncedMemory>data\_;//data数据，指向SyncedMemory的智能指针

shared\_ptr<SyncedMemory>diff\_;//表示“差”，用于更新data\_

**int**num\_;

**int**channels\_;

**int**height\_;

**int**width\_;

**int**count\_;

构造函数

**Blob**():num\_(0),channels\_(0),height\_(0),width\_(0),count\_(0),data\_(),diff\_(){}

功能：简单的初始化

**explicitBlob**(**constint**num,**constint**channels,**constint**height,**constint**width);

功能：调用Reshape函数，初始化数据成员

**template**<**typenameDtype**>

**Blob<Dtype>::Blob**(**constint**num,**constint**channels,**constint**height,

**constint**width) {

Reshape(num,channels, height, width);

}

析构函数

**virtual~Blob**(){}

功能：啥都没做？

**voidReshape**(**constint**num,**constint**height,

**constint**width,**constint**channels);

功能：初始化数据成员，智能指针指向SyncedMemory对象。此时SyncedMemory对象其实并没有为自己的“数据”申请内存，只是自己“数据”的大小（size）。

**template**<**typenameDtype**>

**voidBlob<Dtype>::Reshape**(**constint**num,**constint**channels,**constint**height,

**constint**width) {

CHECK\_GE(num,0);

CHECK\_GE(channels,0);

CHECK\_GE(height,0);

CHECK\_GE(width,0);

num\_= num;

channels\_= channels;

height\_= height;

width\_= width;

count\_=num\_\*channels\_\*height\_\*width\_;

**if**(count\_){

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

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

}**else**{

data\_.reset(**reinterpret\_cast**<SyncedMemory\*>(NULL));

diff\_.reset(**reinterpret\_cast**<SyncedMemory\*>(NULL));

}

}

成员访问函数

功能：就是返回一些成员变量

**inlineintnum**()**const**{**return**num\_;}

**inlineintchannels**()**const**{**return**channels\_;}

**inlineintheight**()**const**{**return**height\_;}

**inlineintwidth**()**const**{**return**width\_;}

**inlineintcount**()**const**{**return**count\_;}

**inlineintoffset**(**constint**n,**constint**c = 0, **constint**h = 0,**constint**w = 0) **const**{

**return**((n \* channels\_+ c) \*height\_+ h) \*width\_+ w;

//计算偏移量，因为数据在内存是一维数组形式的，所以需要计算偏移量来访问

}

“数据”指针返回函数

功能：其实这些函数就是调用SyncedMemory的函数，来返回数据的指针

**constDtype**\***cpu\_data**()**const**;

**constDtype**\***gpu\_data**()**const**;

**constDtype**\***cpu\_diff**()**const**;

**constDtype**\***gpu\_diff**()**const**;

**Dtype**\***mutable\_cpu\_data**();

**Dtype**\***mutable\_gpu\_data**();

**Dtype**\***mutable\_cpu\_diff**();

**Dtype**\***mutable\_gpu\_diff**();

**inlineDtypedata\_at**(**constint**n,**constint**c,**constint**h,

**constint**w)**const**{

//从cpu访问数据data

**return**\*(cpu\_data()+ offset(n, c, h, w));

}

**inlineDtypediff\_at**(**constint**n,**constint**c,**constint**h,

**constint**w)**const**{

//从cpu访问数据diff

**return**\*(cpu\_diff() + offset(n, c, h, w));

}

函数**voidUpdate**()

功能：更新data\_的数据，就是减去diff\_的数据。

**template**<**typenameDtype**>

**voidBlob<Dtype>::Update**(){

//We will perform update based on where the data is located.

**switch**(data\_->head()){

**case**SyncedMemory::*HEAD\_AT\_CPU*:

//perform computation on CPU

caffe\_axpy<**Dtype**>(count\_,**Dtype**(-1),

**reinterpret\_cast**<**constDtype**\*>(diff\_->cpu\_data()),

**reinterpret\_cast**<**Dtype**\*>(data\_->mutable\_cpu\_data()));

//在math\_functions.cpp可以找到该函数的实现，其实这函数也是封装了mkl的函数。这里调用是为了实现了两个向量的减法。

**break**;

**case**SyncedMemory::*HEAD\_AT\_GPU*:

**case**SyncedMemory::*SYNCED*:

//perform computation on GPU

caffe\_gpu\_axpy<**Dtype**>(count\_,**Dtype**(-1),

**reinterpret\_cast**<**constDtype**\*>(diff\_->gpu\_data()),

**reinterpret\_cast**<**Dtype**\*>(data\_->mutable\_gpu\_data()));

//在math\_functions.cpp可以找到该函数的实现，其实这函数也是封装了cublas的函数。这里调用是为了实现了两个向量的减法。

**break**;

**default**:

LOG(FATAL)<<"Syncedmemnot initialized.";

}

}

函数**voidCopyFrom**(**const**Blob<**Dtype**>&source,**bool**copy\_diff = **false**,**bool**reshape =**false**);

功能：从source拷贝数据。copy\_diff作为标志来区分是拷贝data还是拷贝diff。

**template**<**typenameDtype**>

**voidBlob<Dtype>::CopyFrom**(**const**Blob&source,**bool**copy\_diff,**bool**reshape) {

**if**(num\_!= source.num() || channels\_!= source.channels() ||

height\_!= source.height() || width\_!= source.width()) {

**if**(reshape) {

Reshape(source.num(),source.channels(), source.height(), source.width());

}**else**{

LOG(FATAL)<<"Tryingto copy blobs of different sizes.";

}

}

**switch**(Caffe::*mode*()){

**case**Caffe::*GPU*:

**if**(copy\_diff){

CUDA\_CHECK(cudaMemcpy(diff\_->mutable\_gpu\_data(),source.gpu\_diff(),

**sizeof**(**Dtype**)\*count\_,cudaMemcpyDeviceToDevice));

}**else**{

CUDA\_CHECK(cudaMemcpy(data\_->mutable\_gpu\_data(),source.gpu\_data(),

**sizeof**(**Dtype**)\*count\_,cudaMemcpyDeviceToDevice));

}

**break**;

**case**Caffe::*CPU*:

**if**(copy\_diff){

memcpy(diff\_->mutable\_cpu\_data(),source.cpu\_diff(),

**sizeof**(**Dtype**)\*count\_);

}**else**{

memcpy(data\_->mutable\_cpu\_data(),source.cpu\_data(),

**sizeof**(**Dtype**)\*count\_);

}

**break**;

**default**:

LOG(FATAL)<<"Unknowncaffemode.";

}

}

函数**voidFromProto**(**const**BlobProto&proto);

功能：从proto读数据进来，其实就是反序列化

**template**<**typenameDtype**>

**voidBlob<Dtype>::FromProto**(**const**BlobProto&proto){

Reshape(proto.num(),proto.channels(),proto.height(),proto.width());

//copy data

**Dtype**\*data\_vec = mutable\_cpu\_data();

**for**(**int**i = 0; i < count\_;++i) {

data\_vec[i]=proto.data(i);

}

**if**(proto.diff\_size()> 0) {

**Dtype**\*diff\_vec = mutable\_cpu\_diff();

**for**(**int**i = 0; i < count\_;++i) {

diff\_vec[i]=proto.diff(i);

}

}

}

函数**voidToProto**(BlobProto\*proto,**bool**write\_diff = **false**)**const**;

功能：序列化到proto保存

**template**<**typenameDtype**>

**voidBlob<Dtype>::ToProto**(BlobProto\*proto,**bool**write\_diff)**const**{

proto->set\_num(num\_);

proto->set\_channels(channels\_);

proto->set\_height(height\_);

proto->set\_width(width\_);

proto->clear\_data();

proto->clear\_diff();

**constDtype**\*data\_vec = cpu\_data();

**for**(**int**i = 0; i < count\_;++i) {

proto->add\_data(data\_vec[i]);

}

**if**(write\_diff) {

**constDtype**\*diff\_vec = cpu\_diff();

**for**(**int**i = 0; i < count\_;++i) {

proto->add\_diff(diff\_vec[i]);

}

}

}

# [caffe源码分析--data\_layer.cpp](http://blog.csdn.net/lingerlanlan/article/details/27348265)

dataLayer作为整个网络的输入层，

数据从leveldb中取。leveldb的数据是通过图片转换过来的。

网络建立的时候，

datalayer主要是负责设置一些参数，比如batchsize，channels，height，width等。

这次会通过读leveldb一个数据块来获取这些信息。

然后启动一个线程来预先从leveldb拉取一批数据，这些数据是图像数据和图像标签。

正向传播的时候，

datalayer就把预先拉取好数据拷贝到指定的cpu或者gpu的内存。

然后启动新线程再预先拉取数据，这些数据留到下一次正向传播使用。

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/27348265)[在CODE上查看代码片](https://code.csdn.net/snippets/371885)

1. // Copyright 2013 Yangqing Jia
3. #include <stdint.h>
4. #include <leveldb/db.h>
5. #include <pthread.h>
7. #include <string>
8. #include <vector>
10. #include "caffe/layer.hpp"
11. #include "caffe/util/io.hpp"
12. #include "caffe/vision\_layers.hpp"
14. **using** std::string;
16. **namespace** caffe {
18. **template** <**typename** Dtype>
19. **void**\* DataLayerPrefetch(**void**\* layer\_pointer) {
20. CHECK(layer\_pointer);
21. DataLayer<Dtype>\* layer = **reinterpret\_cast**<DataLayer<Dtype>\*>(layer\_pointer);
22. CHECK(layer);
23. Datum datum;
24. CHECK(layer->prefetch\_data\_);
25. Dtype\* top\_data = layer->prefetch\_data\_->mutable\_cpu\_data();//数据
26. Dtype\* top\_label = layer->prefetch\_label\_->mutable\_cpu\_data();//标签
27. **const** Dtype scale = layer->layer\_param\_.scale();
28. **const** **int** batchsize = layer->layer\_param\_.batchsize();
29. **const** **int** cropsize = layer->layer\_param\_.cropsize();
30. **const** **bool** mirror = layer->layer\_param\_.mirror();
32. **if** (mirror && cropsize == 0) {//当前实现需要同时设置mirror和cropsize
33. LOG(FATAL) << "Current implementation requires mirror and cropsize to be "
34. << "set at the same time.";
35. }
36. // datum scales
37. **const** **int** channels = layer->datum\_channels\_;
38. **const** **int** height = layer->datum\_height\_;
39. **const** **int** width = layer->datum\_width\_;
40. **const** **int** size = layer->datum\_size\_;
41. **const** Dtype\* mean = layer->data\_mean\_.cpu\_data();
42. **for** (**int** itemid = 0; itemid < batchsize; ++itemid) {//每一批数据的数量是batchsize，一个循环拉取一张？
43. // get a blob
44. CHECK(layer->iter\_);
45. CHECK(layer->iter\_->Valid());
46. datum.ParseFromString(layer->iter\_->value().ToString());//利用迭代器拉取下一批数据
47. **const** string& data = datum.data();
48. **if** (cropsize) {//如果需要裁剪
49. CHECK(data.size()) << "Image cropping only support uint8 data";
50. **int** h\_off, w\_off;
51. // We only do random crop when we do training.
52. //只是在训练阶段做随机裁剪
53. **if** (Caffe::phase() == Caffe::TRAIN) {
54. // NOLINT\_NEXT\_LINE(runtime/threadsafe\_fn)
55. h\_off = rand() % (height - cropsize);
56. // NOLINT\_NEXT\_LINE(runtime/threadsafe\_fn)
57. w\_off = rand() % (width - cropsize);
58. } **else** {//测试阶段固定裁剪
59. h\_off = (height - cropsize) / 2;
60. w\_off = (width - cropsize) / 2;
61. }
62. // NOLINT\_NEXT\_LINE(runtime/threadsafe\_fn)
63. //怎么感觉下面两种情况的代码是一样的？
64. **if** (mirror && rand() % 2) {
65. // Copy mirrored version
66. **for** (**int** c = 0; c < channels; ++c) {
67. **for** (**int** h = 0; h < cropsize; ++h) {
68. **for** (**int** w = 0; w < cropsize; ++w) {
69. top\_data[((itemid \* channels + c) \* cropsize + h) \* cropsize
70. + cropsize - 1 - w] =
71. (**static\_cast**<Dtype>(
72. (uint8\_t)data[(c \* height + h + h\_off) \* width
73. + w + w\_off])
74. - mean[(c \* height + h + h\_off) \* width + w + w\_off])
75. \* scale;
76. }
77. }
78. }
79. } **else** {
80. // Normal copy
81. **for** (**int** c = 0; c < channels; ++c) {
82. **for** (**int** h = 0; h < cropsize; ++h) {
83. **for** (**int** w = 0; w < cropsize; ++w) {
84. top\_data[((itemid \* channels + c) \* cropsize + h) \* cropsize + w]
85. = (**static\_cast**<Dtype>(
86. (uint8\_t)data[(c \* height + h + h\_off) \* width
87. + w + w\_off])
88. - mean[(c \* height + h + h\_off) \* width + w + w\_off])
89. \* scale;
90. }
91. }
92. }
93. }
94. } **else** {//如果不需要裁剪
95. // we will prefer to use data() first, and then try float\_data()
96. //我们优先考虑data()，然后float\_data()
97. **if** (data.size()) {
98. **for** (**int** j = 0; j < size; ++j) {
99. top\_data[itemid \* size + j] =
100. (**static\_cast**<Dtype>((uint8\_t)data[j]) - mean[j]) \* scale;
101. }
102. } **else** {
103. **for** (**int** j = 0; j < size; ++j) {
104. top\_data[itemid \* size + j] =
105. (datum.float\_data(j) - mean[j]) \* scale;
106. }
107. }
108. }
110. top\_label[itemid] = datum.label();
111. // go to the next iter
112. layer->iter\_->Next();
113. **if** (!layer->iter\_->Valid()) {
114. // We have reached the end. Restart from the first.
115. DLOG(INFO) << "Restarting data prefetching from start.";
116. layer->iter\_->SeekToFirst();
117. }
118. }
120. **return** **reinterpret\_cast**<**void**\*>(NULL);
121. }
123. **template** <**typename** Dtype>
124. DataLayer<Dtype>::~DataLayer<Dtype>() {
125. // Finally, join the thread
126. CHECK(!pthread\_join(thread\_, NULL)) << "Pthread joining failed.";
127. }
129. **template** <**typename** Dtype>
130. **void** DataLayer<Dtype>::SetUp(**const** vector<Blob<Dtype>\*>& bottom,
131. vector<Blob<Dtype>\*>\* top) {
132. CHECK\_EQ(bottom.size(), 0) << "Data Layer takes no input blobs.";
133. CHECK\_EQ(top->size(), 2) << "Data Layer takes two blobs as output.";
134. // Initialize the leveldb
135. leveldb::DB\* db\_temp;
136. leveldb::Options options;
137. options.create\_if\_missing = **false**;
138. options.max\_open\_files = 100;
139. LOG(INFO) << "Opening leveldb " << **this**->layer\_param\_.source();
140. leveldb::Status status = leveldb::DB::Open(
141. options, **this**->layer\_param\_.source(), &db\_temp);
142. CHECK(status.ok()) << "Failed to open leveldb "
143. << **this**->layer\_param\_.source() << std::endl << status.ToString();
144. db\_.reset(db\_temp);
145. iter\_.reset(db\_->NewIterator(leveldb::ReadOptions()));//通过迭代器来操纵leveldb
146. iter\_->SeekToFirst();
147. // Check if we would need to randomly skip a few data points
148. //是否要随机跳过一些数据
149. **if** (**this**->layer\_param\_.rand\_skip()) {
150. // NOLINT\_NEXT\_LINE(runtime/threadsafe\_fn)
151. unsigned **int** skip = rand() % **this**->layer\_param\_.rand\_skip();
152. LOG(INFO) << "Skipping first " << skip << " data points.";
153. **while** (skip-- > 0) {//循环次数
154. iter\_->Next();
155. **if** (!iter\_->Valid()) {
156. iter\_->SeekToFirst();
157. }
158. }
159. }
160. // Read a data point, and use it to initialize the top blob.
161. //读取一个数据点，用来初始化topblob。所谓初始化，只要是指reshape。
162. //可以观察到下面iter\_调用调用next。所以这次读取只是用来读取出来channels等参数的，不作处理。
163. Datum datum;
164. datum.ParseFromString(iter\_->value().ToString());//利用迭代器读取第一个数据点
165. // image图像数据
166. **int** cropsize = **this**->layer\_param\_.cropsize();//裁剪大小
167. **if** (cropsize > 0) {//需要裁剪
168. (\*top)[0]->Reshape(
169. **this**->layer\_param\_.batchsize(), datum.channels(), cropsize, cropsize);
170. prefetch\_data\_.reset(**new** Blob<Dtype>(
171. **this**->layer\_param\_.batchsize(), datum.channels(), cropsize, cropsize));
172. } **else** {//不需要裁剪
173. (\*top)[0]->Reshape(
174. **this**->layer\_param\_.batchsize(), datum.channels(), datum.height(),
175. datum.width());
176. prefetch\_data\_.reset(**new** Blob<Dtype>(
177. **this**->layer\_param\_.batchsize(), datum.channels(), datum.height(),
178. datum.width()));
179. }
180. LOG(INFO) << "output data size: " << (\*top)[0]->num() << ","
181. << (\*top)[0]->channels() << "," << (\*top)[0]->height() << ","
182. << (\*top)[0]->width();
183. // label标签数据
184. (\*top)[1]->Reshape(**this**->layer\_param\_.batchsize(), 1, 1, 1);
185. prefetch\_label\_.reset(
186. **new** Blob<Dtype>(**this**->layer\_param\_.batchsize(), 1, 1, 1));
187. // datum size
188. datum\_channels\_ = datum.channels();
189. datum\_height\_ = datum.height();
190. datum\_width\_ = datum.width();
191. datum\_size\_ = datum.channels() \* datum.height() \* datum.width();
192. CHECK\_GT(datum\_height\_, cropsize);
193. CHECK\_GT(datum\_width\_, cropsize);
194. // check if we want to have mean是否要减去均值
195. **if** (**this**->layer\_param\_.has\_meanfile()) {
196. BlobProto blob\_proto;
197. LOG(INFO) << "Loading mean file from" << **this**->layer\_param\_.meanfile();
198. ReadProtoFromBinaryFile(**this**->layer\_param\_.meanfile().c\_str(), &blob\_proto);
199. data\_mean\_.FromProto(blob\_proto);
200. CHECK\_EQ(data\_mean\_.num(), 1);
201. CHECK\_EQ(data\_mean\_.channels(), datum\_channels\_);
202. CHECK\_EQ(data\_mean\_.height(), datum\_height\_);
203. CHECK\_EQ(data\_mean\_.width(), datum\_width\_);
204. } **else** {
205. // Simply initialize an all-empty mean.
206. data\_mean\_.Reshape(1, datum\_channels\_, datum\_height\_, datum\_width\_);
207. }
208. // Now, start the prefetch thread. Before calling prefetch, we make two
209. // cpu\_data calls so that the prefetch thread does not accidentally make
210. // simultaneous cudaMalloc calls when the main thread is running. In some
211. // GPUs this seems to cause failures if we do not so.
212. prefetch\_data\_->mutable\_cpu\_data();
213. prefetch\_label\_->mutable\_cpu\_data();
214. data\_mean\_.cpu\_data();
215. DLOG(INFO) << "Initializing prefetch";
216. CHECK(!pthread\_create(&thread\_, NULL, DataLayerPrefetch<Dtype>,
217. **reinterpret\_cast**<**void**\*>(**this**))) << "Pthread execution failed.";
218. DLOG(INFO) << "Prefetch initialized.";
219. }
221. **template** <**typename** Dtype>
222. **void** DataLayer<Dtype>::Forward\_cpu(**const** vector<Blob<Dtype>\*>& bottom,
223. vector<Blob<Dtype>\*>\* top) {
224. // First, join the thread 等待线程结束
225. CHECK(!pthread\_join(thread\_, NULL)) << "Pthread joining failed.";
226. // Copy the data拷贝数据到top，即该层的输出
227. memcpy((\*top)[0]->mutable\_cpu\_data(), prefetch\_data\_->cpu\_data(),
228. **sizeof**(Dtype) \* prefetch\_data\_->count());
229. memcpy((\*top)[1]->mutable\_cpu\_data(), prefetch\_label\_->cpu\_data(),
230. **sizeof**(Dtype) \* prefetch\_label\_->count());
231. // Start a new prefetch thread启动新线程拉取下一批数据
232. CHECK(!pthread\_create(&thread\_, NULL, DataLayerPrefetch<Dtype>,
233. **reinterpret\_cast**<**void**\*>(**this**))) << "Pthread execution failed.";
234. }
236. // The backward operations are dummy - they do not carry any computation.
237. **template** <**typename** Dtype>
238. Dtype DataLayer<Dtype>::Backward\_cpu(**const** vector<Blob<Dtype>\*>& top,
239. **const** **bool** propagate\_down, vector<Blob<Dtype>\*>\* bottom) {
240. **return** Dtype(0.);
241. }
243. INSTANTIATE\_CLASS(DataLayer);
245. }  // namespace caffe

[caffe源码分析--poolinger\_layer.cpp](http://blog.csdn.net/lingerlanlan/article/details/38294169)

对于采样层，cafffe里实现了最大采样和平均采样的算法。

最大采样，给定一个扫描窗口，找最大值，

平均采样，扫描窗口内所有值的平均值。

其实对于caffe的实现一直有个疑问，

就是每一层貌似没有绑定一个激活函数？

看ufldl教程，感觉激活函数是必要存在的。

这怎么解释呢？

看到源码中，看到一些激活函数，比如sigmoid\_layer.cpp和sigmoid\_layer.cu。

也就是说，激活函数作为layer层面来实现了。当然，还有tanh\_layer和relu\_layer。

那，这个意思是说，让我们建立网络的时候更加随意，可自由搭配激活函数吗？

但是，我看了caffe自带的那些例子，貌似很少见到用了激活函数layer的，顶多看到用了relu\_layer，其他的没见过。

这意思是说，激活函数不重要吗？真是费解啊。

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/38294169)

1. // Copyright 2013 Yangqing Jia
3. #include <algorithm>
4. #include <cfloat>
5. #include <vector>
7. #include "caffe/layer.hpp"
8. #include "caffe/vision\_layers.hpp"
9. #include "caffe/util/math\_functions.hpp"
11. **using** std::max;
12. **using** std::min;
14. **namespace** caffe {
16. **template** <**typename** Dtype>
17. **void** PoolingLayer<Dtype>::SetUp(**const** vector<Blob<Dtype>\*>& bottom,
18. vector<Blob<Dtype>\*>\* top) {
19. CHECK\_EQ(bottom.size(), 1) << "PoolingLayer takes a single blob as input.";
20. CHECK\_EQ(top->size(), 1) << "PoolingLayer takes a single blob as output.";
21. KSIZE\_ = **this**->layer\_param\_.kernelsize();//核大小
22. STRIDE\_ = **this**->layer\_param\_.stride();//步长
23. CHANNELS\_ = bottom[0]->channels();//通道
24. HEIGHT\_ = bottom[0]->height();//高
25. WIDTH\_ = bottom[0]->width();//宽
26. POOLED\_HEIGHT\_ = **static\_cast**<**int**>(
27. ceil(**static\_cast**<**float**>(HEIGHT\_ - KSIZE\_) / STRIDE\_)) + 1;//计算采样之后的高
28. POOLED\_WIDTH\_ = **static\_cast**<**int**>(
29. ceil(**static\_cast**<**float**>(WIDTH\_ - KSIZE\_) / STRIDE\_)) + 1;//计算采样之后的宽
30. (\*top)[0]->Reshape(bottom[0]->num(), CHANNELS\_, POOLED\_HEIGHT\_,//采样之后大小
31. POOLED\_WIDTH\_);
32. // If stochastic pooling, we will initialize the random index part.
33. **if** (**this**->layer\_param\_.pool() == LayerParameter\_PoolMethod\_STOCHASTIC) {
34. rand\_idx\_.Reshape(bottom[0]->num(), CHANNELS\_, POOLED\_HEIGHT\_,
35. POOLED\_WIDTH\_);
36. }
37. }
39. // TODO(Yangqing): Is there a faster way to do pooling in the channel-first
40. // case?
41. **template** <**typename** Dtype>
42. **void** PoolingLayer<Dtype>::Forward\_cpu(**const** vector<Blob<Dtype>\*>& bottom,
43. vector<Blob<Dtype>\*>\* top) {
44. **const** Dtype\* bottom\_data = bottom[0]->cpu\_data();//采样层输入
45. Dtype\* top\_data = (\*top)[0]->mutable\_cpu\_data();//采样层输出
46. // Different pooling methods. We explicitly do the switch outside the for
47. // loop to save time, although this results in more codes.
48. **int** top\_count = (\*top)[0]->count();
49. **switch** (**this**->layer\_param\_.pool()) {
50. **case** LayerParameter\_PoolMethod\_MAX://最大采样方法
51. // Initialize
52. **for** (**int** i = 0; i < top\_count; ++i) {
53. top\_data[i] = -FLT\_MAX;
54. }
55. // The main loop
56. **for** (**int** n = 0; n < bottom[0]->num(); ++n) {
57. **for** (**int** c = 0; c < CHANNELS\_; ++c) {
58. **for** (**int** ph = 0; ph < POOLED\_HEIGHT\_; ++ph) {
59. **for** (**int** pw = 0; pw < POOLED\_WIDTH\_; ++pw) {
60. **int** hstart = ph \* STRIDE\_;
61. **int** wstart = pw \* STRIDE\_;
62. **int** hend = min(hstart + KSIZE\_, HEIGHT\_);
63. **int** wend = min(wstart + KSIZE\_, WIDTH\_);
64. **for** (**int** h = hstart; h < hend; ++h) {//找出核范围内最大
65. **for** (**int** w = wstart; w < wend; ++w) {
66. top\_data[ph \* POOLED\_WIDTH\_ + pw] =
67. max(top\_data[ph \* POOLED\_WIDTH\_ + pw],
68. bottom\_data[h \* WIDTH\_ + w]);
69. }
70. }
71. }
72. }
73. // compute offset 指针移动到下一个channel。注意代码这里的位置。采样是针对每个channel的。
74. bottom\_data += bottom[0]->offset(0, 1);
75. top\_data += (\*top)[0]->offset(0, 1);
76. }
77. }
78. **break**;
79. **case** LayerParameter\_PoolMethod\_AVE:
80. **for** (**int** i = 0; i < top\_count; ++i) {
81. top\_data[i] = 0;
82. }
83. // The main loop
84. **for** (**int** n = 0; n < bottom[0]->num(); ++n) {
85. **for** (**int** c = 0; c < CHANNELS\_; ++c) {
86. **for** (**int** ph = 0; ph < POOLED\_HEIGHT\_; ++ph) {
87. **for** (**int** pw = 0; pw < POOLED\_WIDTH\_; ++pw) {
88. **int** hstart = ph \* STRIDE\_;
89. **int** wstart = pw \* STRIDE\_;
90. **int** hend = min(hstart + KSIZE\_, HEIGHT\_);
91. **int** wend = min(wstart + KSIZE\_, WIDTH\_);
92. **for** (**int** h = hstart; h < hend; ++h) {//核范围内算平均
93. **for** (**int** w = wstart; w < wend; ++w) {
94. top\_data[ph \* POOLED\_WIDTH\_ + pw] +=
95. bottom\_data[h \* WIDTH\_ + w];
96. }
97. }
98. top\_data[ph \* POOLED\_WIDTH\_ + pw] /=
99. (hend - hstart) \* (wend - wstart);
100. }
101. }
102. // compute offset
103. bottom\_data += bottom[0]->offset(0, 1);
104. top\_data += (\*top)[0]->offset(0, 1);
105. }
106. }
107. **break**;
108. **case** LayerParameter\_PoolMethod\_STOCHASTIC://这种算法这里未实现
109. NOT\_IMPLEMENTED;
110. **break**;
111. **default**:
112. LOG(FATAL) << "Unknown pooling method.";
113. }
114. }
116. **template** <**typename** Dtype>
117. Dtype PoolingLayer<Dtype>::Backward\_cpu(**const** vector<Blob<Dtype>\*>& top,
118. **const** **bool** propagate\_down, vector<Blob<Dtype>\*>\* bottom) {
119. **if** (!propagate\_down) {
120. **return** Dtype(0.);
121. }
122. **const** Dtype\* top\_diff = top[0]->cpu\_diff();
123. **const** Dtype\* top\_data = top[0]->cpu\_data();
124. **const** Dtype\* bottom\_data = (\*bottom)[0]->cpu\_data();
125. Dtype\* bottom\_diff = (\*bottom)[0]->mutable\_cpu\_diff();
126. // Different pooling methods. We explicitly do the switch outside the for
127. // loop to save time, although this results in more codes.
128. memset(bottom\_diff, 0, (\*bottom)[0]->count() \* **sizeof**(Dtype));
129. **switch** (**this**->layer\_param\_.pool()) {
130. **case** LayerParameter\_PoolMethod\_MAX:
131. // The main loop
132. **for** (**int** n = 0; n < top[0]->num(); ++n) {
133. **for** (**int** c = 0; c < CHANNELS\_; ++c) {
134. **for** (**int** ph = 0; ph < POOLED\_HEIGHT\_; ++ph) {
135. **for** (**int** pw = 0; pw < POOLED\_WIDTH\_; ++pw) {
136. **int** hstart = ph \* STRIDE\_;
137. **int** wstart = pw \* STRIDE\_;
138. **int** hend = min(hstart + KSIZE\_, HEIGHT\_);
139. **int** wend = min(wstart + KSIZE\_, WIDTH\_);
140. **for** (**int** h = hstart; h < hend; ++h) {
141. **for** (**int** w = wstart; w < wend; ++w) {
142. bottom\_diff[h \* WIDTH\_ + w] +=//采样层输出的残传播给输入。由于是最大采样方法，输出存的都是输入范围内最大的值，所以残差传播的时候也只有范围内最大的值受影响
143. top\_diff[ph \* POOLED\_WIDTH\_ + pw] \*
144. (bottom\_data[h \* WIDTH\_ + w] ==
145. top\_data[ph \* POOLED\_WIDTH\_ + pw]);
146. }
147. }
148. }
149. }
150. // offset  移动到下一个channel
151. bottom\_data += (\*bottom)[0]->offset(0, 1);
152. top\_data += top[0]->offset(0, 1);
153. bottom\_diff += (\*bottom)[0]->offset(0, 1);
154. top\_diff += top[0]->offset(0, 1);
155. }
156. }
157. **break**;
158. **case** LayerParameter\_PoolMethod\_AVE:
159. // The main loop
160. **for** (**int** n = 0; n < top[0]->num(); ++n) {
161. **for** (**int** c = 0; c < CHANNELS\_; ++c) {
162. **for** (**int** ph = 0; ph < POOLED\_HEIGHT\_; ++ph) {
163. **for** (**int** pw = 0; pw < POOLED\_WIDTH\_; ++pw) {
164. **int** hstart = ph \* STRIDE\_;
165. **int** wstart = pw \* STRIDE\_;
166. **int** hend = min(hstart + KSIZE\_, HEIGHT\_);
167. **int** wend = min(wstart + KSIZE\_, WIDTH\_);
168. **int** poolsize = (hend - hstart) \* (wend - wstart);
169. **for** (**int** h = hstart; h < hend; ++h) {
170. **for** (**int** w = wstart; w < wend; ++w) {
171. bottom\_diff[h \* WIDTH\_ + w] +=//采样层输出的残差传播给输入，由于是平均采样，所以权重都是1 / poolsize。
172. top\_diff[ph \* POOLED\_WIDTH\_ + pw] / poolsize;
173. }
174. }
175. }
176. }
177. // offset
178. bottom\_data += (\*bottom)[0]->offset(0, 1);
179. top\_data += top[0]->offset(0, 1);
180. bottom\_diff += (\*bottom)[0]->offset(0, 1);
181. top\_diff += top[0]->offset(0, 1);
182. }
183. }
184. **break**;
185. **case** LayerParameter\_PoolMethod\_STOCHASTIC:
186. NOT\_IMPLEMENTED;
187. **break**;
188. **default**:
189. LOG(FATAL) << "Unknown pooling method.";
190. }
191. **return** Dtype(0.);
192. }

195. INSTANTIATE\_CLASS(PoolingLayer);

198. }  // namespace caffe

[caffe源码修改：抽取任意一张图片的特征](http://blog.csdn.net/lingerlanlan/article/details/39400375)

目前caffe不是很完善，输入的图片数据需要在prototxt指定路径。但是我们往往有这么一个需求：训练后得到一个模型文件，我们想拿这个模型文件来对一张图片抽取特征或者预测分类等。如果非得在prototxt指定路径，就很不方便。因此，这样的工具才是我们需要的：给一个可执行文件通过命令行来传递图片路径，然后caffe读入图片数据，进行一次正向传播。

因此我做了这么一个工具，用来抽取任意一张图片的特征。

这工具的使用方法如下：

extract\_one\_feature.bin ./model/caffe\_reference\_imagenet\_model ./examples/\_temp/imagenet\_val.prototxt fc7 ./examples/\_temp/features /media/G/imageset/clothing/针织衫/针织衫\_426.jpg CPU

参数1：./model/caffe\_reference\_imagenet\_model是训练后的模型文件

参数2：./examples/\_temp/imagenet\_val.prototxt 网络配置文件

参数3：fc7是blob的名字

参数4：./examples/\_temp/features 将该图片的特征保存在该文件

参数5：图片路径

参数6：GPU或者CPU模式

（其实我还想到更好的工具，如果该可执行文件是监听模式的，就是通过一定的方式，给该进程传递 图片路径，进程接到任务就执行。

这样子的话，就不需要每次抽一张图片都要申请内存空间。(\*^\_\_^\*) 嘻嘻……）

下面给出初步修改方法，大家可以根据自己需求再修改。

extract\_one\_feature.cpp（该文件参考过源码中extract\_features.cpp修改）

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/39400375)[在CODE上查看代码片](https://code.csdn.net/snippets/471699)

1. #include <stdio.h>  // for snprintf
2. #include <string>
3. #include <vector>
4. #include <iostream>
5. #include <fstream>
7. #include "boost/algorithm/string.hpp"
8. #include "google/protobuf/text\_format.h"
9. #include "leveldb/db.h"
10. #include "leveldb/write\_batch.h"
12. #include "caffe/blob.hpp"
13. #include "caffe/common.hpp"
14. #include "caffe/net.hpp"
15. #include "caffe/proto/caffe.pb.h"
16. #include "caffe/util/io.hpp"
17. #include "caffe/vision\_layers.hpp"
19. **using** **namespace** caffe;  // NOLINT(build/namespaces)
21. **template**<**typename** Dtype>
22. **int** feature\_extraction\_pipeline(**int** argc, **char**\*\* argv);
24. **int** main(**int** argc, **char**\*\* argv) {
25. **return** feature\_extraction\_pipeline<**float**>(argc, argv);
26. //  return feature\_extraction\_pipeline<double>(argc, argv);
27. }
29. **template**<**typename** Dtype>
30. **class** writeDb
31. {
32. **public**:
33. **void** open(string dbName)
34. {
35. db.open(dbName.c\_str());
36. }
37. **void** write(**const** Dtype &data)
38. {
39. db<<data;
40. }
41. **void** write(**const** string &str)
42. {
43. db<<str;
44. }
45. **virtual** ~writeDb()
46. {
47. db.close();
48. }
49. **private**:
50. std::ofstream db;
51. };
53. **template**<**typename** Dtype>
54. **int** feature\_extraction\_pipeline(**int** argc, **char**\*\* argv) {
55. ::google::InitGoogleLogging(argv[0]);
56. **const** **int** num\_required\_args = 6;
57. **if** (argc < num\_required\_args) {
58. LOG(ERROR)<<
59. "This program takes in a trained network and an input data layer, and then"
60. " extract features of the input data produced by the net.\n"
61. "Usage: extract\_features  pretrained\_net\_param"
62. "  feature\_extraction\_proto\_file  extract\_feature\_blob\_name1[,name2,...]"
63. "  save\_feature\_leveldb\_name1[,name2,...]  img\_path  [CPU/GPU]"
64. "  [DEVICE\_ID=0]\n"
65. "Note: you can extract multiple features in one pass by specifying"
66. " multiple feature blob names and leveldb names seperated by ','."
67. " The names cannot contain white space characters and the number of blobs"
68. " and leveldbs must be equal.";
69. **return** 1;
70. }
71. **int** arg\_pos = num\_required\_args;
73. arg\_pos = num\_required\_args;
74. **if** (argc > arg\_pos && strcmp(argv[arg\_pos], "GPU") == 0) {
75. LOG(ERROR)<< "Using GPU";
76. uint device\_id = 0;
77. **if** (argc > arg\_pos + 1) {
78. device\_id = atoi(argv[arg\_pos + 1]);
79. CHECK\_GE(device\_id, 0);
80. }
81. LOG(ERROR) << "Using Device\_id=" << device\_id;
82. Caffe::SetDevice(device\_id);
83. Caffe::set\_mode(Caffe::GPU);
84. } **else** {
85. LOG(ERROR) << "Using CPU";
86. Caffe::set\_mode(Caffe::CPU);
87. }
88. Caffe::set\_phase(Caffe::TEST);
90. arg\_pos = 0;  // the name of the executable
91. string pretrained\_binary\_proto(argv[++arg\_pos]);//网络模型参数文件
93. string feature\_extraction\_proto(argv[++arg\_pos]);
95. shared\_ptr<Net<Dtype> > feature\_extraction\_net(
96. **new** Net<Dtype>(feature\_extraction\_proto));
98. feature\_extraction\_net->CopyTrainedLayersFrom(pretrained\_binary\_proto);//将网络参数load进内存

101. string extract\_feature\_blob\_names(argv[++arg\_pos]);
102. vector<string> blob\_names;//要抽取特征的layer的名字，可以是多个
103. boost::split(blob\_names, extract\_feature\_blob\_names, boost::is\_any\_of(","));
105. string save\_feature\_leveldb\_names(argv[++arg\_pos]);
106. vector<string> leveldb\_names;// 这里我改写成一个levedb为一个文件，数据格式不使用真正的levedb，而是自定义
107. boost::split(leveldb\_names, save\_feature\_leveldb\_names,
108. boost::is\_any\_of(","));
109. CHECK\_EQ(blob\_names.size(), leveldb\_names.size()) <<
110. " the number of blob names and leveldb names must be equal";
111. **size\_t** num\_features = blob\_names.size();
113. **for** (**size\_t** i = 0; i < num\_features; i++) {
114. CHECK(feature\_extraction\_net->has\_blob(blob\_names[i]))  //检测blob的名字在网络中是否存在
115. << "Unknown feature blob name " << blob\_names[i]
116. << " in the network " << feature\_extraction\_proto;
117. }

120. vector<shared\_ptr<writeDb<Dtype> > > feature\_dbs;
121. **for** (**size\_t** i = 0; i < num\_features; ++i) //打开db，准备写入数据
122. {
123. LOG(INFO)<< "Opening db " << leveldb\_names[i];
124. writeDb<Dtype>\* db = **new** writeDb<Dtype>();
125. db->open(leveldb\_names[i]);
126. feature\_dbs.push\_back(shared\_ptr<writeDb<Dtype> >(db));
127. }


131. LOG(ERROR)<< "Extacting Features";
133. **const** shared\_ptr<Layer<Dtype> > layer = feature\_extraction\_net->layer\_by\_name("data");//获取第一层
134. MyImageDataLayer<Dtype>\* my\_layer = (MyImageDataLayer<Dtype>\*)layer.get();
135. my\_layer->setImgPath(argv[++arg\_pos],1);//"/media/G/imageset/clothing/针织衫/针织衫\_1.jpg"
136. //设置图片路径
138. vector<Blob<**float**>\*> input\_vec;
139. vector<**int**> image\_indices(num\_features, 0);
140. **int** num\_mini\_batches = 1;//atoi(argv[++arg\_pos]);//共多少次迭代。  每次迭代的数量在prototxt用batchsize指定
141. **for** (**int** batch\_index = 0; batch\_index < num\_mini\_batches; ++batch\_index) //共num\_mini\_batches次迭代
142. {
143. feature\_extraction\_net->Forward(input\_vec);//一次正向传播
144. **for** (**int** i = 0; i < num\_features; ++i) //多层特征
145. {
146. **const** shared\_ptr<Blob<Dtype> > feature\_blob = feature\_extraction\_net
147. ->blob\_by\_name(blob\_names[i]);
148. **int** batch\_size = feature\_blob->num();
149. **int** dim\_features = feature\_blob->count() / batch\_size;
151. Dtype\* feature\_blob\_data;
153. **for** (**int** n = 0; n < batch\_size; ++n)
154. {
155. feature\_blob\_data = feature\_blob->mutable\_cpu\_data() +
156. feature\_blob->offset(n);
157. feature\_dbs[i]->write("3 ");
158. **for** (**int** d = 0; d < dim\_features; ++d)
159. {
160. feature\_dbs[i]->write((Dtype)(d+1));
161. feature\_dbs[i]->write(":");
162. feature\_dbs[i]->write(feature\_blob\_data[d]);
163. feature\_dbs[i]->write(" ");
164. }
165. feature\_dbs[i]->write("\n");
167. }  // for (int n = 0; n < batch\_size; ++n)
168. }  // for (int i = 0; i < num\_features; ++i)
169. }  // for (int batch\_index = 0; batch\_index < num\_mini\_batches; ++batch\_index)

172. LOG(ERROR)<< "Successfully extracted the features!";
173. **return** 0;
174. }

my\_data\_layer.cpp（参考image\_data\_layer修改）

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/39400375)[在CODE上查看代码片](https://code.csdn.net/snippets/471699)

1. #include <fstream>  // NOLINT(readability/streams)
2. #include <iostream>  // NOLINT(readability/streams)
3. #include <string>
4. #include <utility>
5. #include <vector>
7. #include "caffe/layer.hpp"
8. #include "caffe/util/io.hpp"
9. #include "caffe/util/math\_functions.hpp"
10. #include "caffe/util/rng.hpp"
11. #include "caffe/vision\_layers.hpp"
13. **namespace** caffe {

16. **template** <**typename** Dtype>
17. MyImageDataLayer<Dtype>::~MyImageDataLayer<Dtype>() {
18. }

21. **template** <**typename** Dtype>
22. **void** MyImageDataLayer<Dtype>::setImgPath(string path,**int** label)
23. {
24. lines\_.clear();
25. lines\_.push\_back(std::make\_pair(path, label));
26. }

29. **template** <**typename** Dtype>
30. **void** MyImageDataLayer<Dtype>::SetUp(**const** vector<Blob<Dtype>\*>& bottom,
31. vector<Blob<Dtype>\*>\* top) {
32. Layer<Dtype>::SetUp(bottom, top);
33. **const** **int** new\_height  = **this**->layer\_param\_.image\_data\_param().new\_height();
34. **const** **int** new\_width  = **this**->layer\_param\_.image\_data\_param().new\_width();
35. CHECK((new\_height == 0 && new\_width == 0) ||
36. (new\_height > 0 && new\_width > 0)) << "Current implementation requires "
37. "new\_height and new\_width to be set at the same time.";
39. /\*
40. \* 因为下面需要随便拿一张图片来初始化blob。
41. \* 因此需要硬盘上有一张图片。
42. \* 1 从prototxt读取一张图片的路径，
43. \* 2 其实也可以在这里将用于初始化的图片路径写死
44. \*/
46. /\*1\*/
47. /\*
48. const string& source = this->layer\_param\_.image\_data\_param().source();
49. LOG(INFO) << "Opening file " << source;
50. std::ifstream infile(source.c\_str());
51. string filename;
52. int label;
53. while (infile >> filename >> label) {
54. lines\_.push\_back(std::make\_pair(filename, label));
55. }
56. \*/
58. /\*2\*/
59. lines\_.push\_back(std::make\_pair("/home/linger/init.jpg",1));
61. //上面1和2代码可以任意用一段
63. lines\_id\_ = 0;
64. // Read a data point, and use it to initialize the top blob. (随便)读取一张图片，来初始化blob
65. Datum datum;
66. CHECK(ReadImageToDatum(lines\_[lines\_id\_].first, lines\_[lines\_id\_].second,
67. new\_height, new\_width, &datum));
68. // image
69. **const** **int** crop\_size = **this**->layer\_param\_.image\_data\_param().crop\_size();
70. **const** **int** batch\_size = 1;//this->layer\_param\_.image\_data\_param().batch\_size();
71. **const** string& mean\_file = **this**->layer\_param\_.image\_data\_param().mean\_file();
72. **if** (crop\_size > 0) {
73. (\*top)[0]->Reshape(batch\_size, datum.channels(), crop\_size, crop\_size);
74. prefetch\_data\_.Reshape(batch\_size, datum.channels(), crop\_size, crop\_size);
75. } **else** {
76. (\*top)[0]->Reshape(batch\_size, datum.channels(), datum.height(),
77. datum.width());
78. prefetch\_data\_.Reshape(batch\_size, datum.channels(), datum.height(),
79. datum.width());
80. }
81. LOG(INFO) << "output data size: " << (\*top)[0]->num() << ","
82. << (\*top)[0]->channels() << "," << (\*top)[0]->height() << ","
83. << (\*top)[0]->width();
84. // label
85. (\*top)[1]->Reshape(batch\_size, 1, 1, 1);
86. prefetch\_label\_.Reshape(batch\_size, 1, 1, 1);
87. // datum size
88. datum\_channels\_ = datum.channels();
89. datum\_height\_ = datum.height();
90. datum\_width\_ = datum.width();
91. datum\_size\_ = datum.channels() \* datum.height() \* datum.width();
92. CHECK\_GT(datum\_height\_, crop\_size);
93. CHECK\_GT(datum\_width\_, crop\_size);
94. // check if we want to have mean
95. **if** (**this**->layer\_param\_.image\_data\_param().has\_mean\_file()) {
96. BlobProto blob\_proto;
97. LOG(INFO) << "Loading mean file from" << mean\_file;
98. ReadProtoFromBinaryFile(mean\_file.c\_str(), &blob\_proto);
99. data\_mean\_.FromProto(blob\_proto);
100. CHECK\_EQ(data\_mean\_.num(), 1);
101. CHECK\_EQ(data\_mean\_.channels(), datum\_channels\_);
102. CHECK\_EQ(data\_mean\_.height(), datum\_height\_);
103. CHECK\_EQ(data\_mean\_.width(), datum\_width\_);
104. } **else** {
105. // Simply initialize an all-empty mean.
106. data\_mean\_.Reshape(1, datum\_channels\_, datum\_height\_, datum\_width\_);
107. }
108. // Now, start the prefetch thread. Before calling prefetch, we make two
109. // cpu\_data calls so that the prefetch thread does not accidentally make
110. // simultaneous cudaMalloc calls when the main thread is running. In some
111. // GPUs this seems to cause failures if we do not so.
112. prefetch\_data\_.mutable\_cpu\_data();
113. prefetch\_label\_.mutable\_cpu\_data();
114. data\_mean\_.cpu\_data();

117. }
119. //--------------------------------下面是读取一张图片数据-----------------------------------------------
120. **template** <**typename** Dtype>
121. **void** MyImageDataLayer<Dtype>::fetchData() {
122. Datum datum;
123. CHECK(prefetch\_data\_.count());
124. Dtype\* top\_data = prefetch\_data\_.mutable\_cpu\_data();
125. Dtype\* top\_label = prefetch\_label\_.mutable\_cpu\_data();
126. ImageDataParameter image\_data\_param = **this**->layer\_param\_.image\_data\_param();
127. **const** Dtype scale = image\_data\_param.scale();//image\_data\_layer相关参数
128. **const** **int** batch\_size = 1;//image\_data\_param.batch\_size(); 这里我们只需要一张图片
130. **const** **int** crop\_size = image\_data\_param.crop\_size();
131. **const** **bool** mirror = image\_data\_param.mirror();
132. **const** **int** new\_height = image\_data\_param.new\_height();
133. **const** **int** new\_width = image\_data\_param.new\_width();
135. **if** (mirror && crop\_size == 0) {
136. LOG(FATAL) << "Current implementation requires mirror and crop\_size to be "
137. << "set at the same time.";
138. }
139. // datum scales
140. **const** **int** channels = datum\_channels\_;
141. **const** **int** height = datum\_height\_;
142. **const** **int** width = datum\_width\_;
143. **const** **int** size = datum\_size\_;
144. **const** **int** lines\_size = lines\_.size();
145. **const** Dtype\* mean = data\_mean\_.cpu\_data();
147. **for** (**int** item\_id = 0; item\_id < batch\_size; ++item\_id) {//读取一图片
148. // get a blob
149. CHECK\_GT(lines\_size, lines\_id\_);
150. **if** (!ReadImageToDatum(lines\_[lines\_id\_].first,
151. lines\_[lines\_id\_].second,
152. new\_height, new\_width, &datum)) {
153. **continue**;
154. }
155. **const** string& data = datum.data();
156. **if** (crop\_size) {
157. CHECK(data.size()) << "Image cropping only support uint8 data";
158. **int** h\_off, w\_off;
159. // We only do random crop when we do training.
160. h\_off = (height - crop\_size) / 2;
161. w\_off = (width - crop\_size) / 2;
163. // Normal copy 正常读取，把裁剪后的图片数据读给top\_data
164. **for** (**int** c = 0; c < channels; ++c) {
165. **for** (**int** h = 0; h < crop\_size; ++h) {
166. **for** (**int** w = 0; w < crop\_size; ++w) {
167. **int** top\_index = ((item\_id \* channels + c) \* crop\_size + h)
168. \* crop\_size + w;
169. **int** data\_index = (c \* height + h + h\_off) \* width + w + w\_off;
170. Dtype datum\_element =
171. **static\_cast**<Dtype>(**static\_cast**<uint8\_t>(data[data\_index]));
172. top\_data[top\_index] = (datum\_element - mean[data\_index]) \* scale;
173. }
174. }
175. }
177. } **else** {
178. // Just copy the whole data 正常读取，把图片数据读给top\_data
179. **if** (data.size()) {
180. **for** (**int** j = 0; j < size; ++j) {
181. Dtype datum\_element =
182. **static\_cast**<Dtype>(**static\_cast**<uint8\_t>(data[j]));
183. top\_data[item\_id \* size + j] = (datum\_element - mean[j]) \* scale;
184. }
185. } **else** {
186. **for** (**int** j = 0; j < size; ++j) {
187. top\_data[item\_id \* size + j] =
188. (datum.float\_data(j) - mean[j]) \* scale;
189. }
190. }
191. }
192. top\_label[item\_id] = datum.label();//读取该图片的标签
194. }
195. }
197. **template** <**typename** Dtype>
198. Dtype MyImageDataLayer<Dtype>::Forward\_cpu(**const** vector<Blob<Dtype>\*>& bottom,
199. vector<Blob<Dtype>\*>\* top) {
201. //更新input
202. fetchData();
204. // Copy the data
205. caffe\_copy(prefetch\_data\_.count(), prefetch\_data\_.cpu\_data(),
206. (\*top)[0]->mutable\_cpu\_data());
207. caffe\_copy(prefetch\_label\_.count(), prefetch\_label\_.cpu\_data(),
208. (\*top)[1]->mutable\_cpu\_data());
210. **return** Dtype(0.);
211. }
213. #ifdef CPU\_ONLY
214. STUB\_GPU\_FORWARD(ImageDataLayer, Forward);
215. #endif
217. INSTANTIATE\_CLASS(MyImageDataLayer);
219. }  // namespace caffe

在data\_layers.hpp添加一下代码，参考ImageDataLayer写的。

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/39400375)[在CODE上查看代码片](https://code.csdn.net/snippets/471699)

1. **template** <**typename** Dtype>
2. **class** MyImageDataLayer : **public** Layer<Dtype>  {
3. **public**:
4. **explicit** MyImageDataLayer(**const** LayerParameter& param)
5. : Layer<Dtype>(param) {}
6. **virtual** ~MyImageDataLayer();
7. **virtual** **void** SetUp(**const** vector<Blob<Dtype>\*>& bottom,
8. vector<Blob<Dtype>\*>\* top);
10. **virtual** **inline** LayerParameter\_LayerType type() **const** {
11. **return** LayerParameter\_LayerType\_MY\_IMAGE\_DATA;
12. }
13. **virtual** **inline** **int** ExactNumBottomBlobs() **const** { **return** 0; }
14. **virtual** **inline** **int** ExactNumTopBlobs() **const** { **return** 2; }
15. **void** fetchData();
16. **void** setImgPath(string path,**int** label);
17. **protected**:
18. **virtual** Dtype Forward\_cpu(**const** vector<Blob<Dtype>\*>& bottom,
19. vector<Blob<Dtype>\*>\* top);
21. **virtual** **void** Backward\_cpu(**const** vector<Blob<Dtype>\*>& top,
22. **const** vector<**bool**>& propagate\_down, vector<Blob<Dtype>\*>\* bottom) {}

25. vector<std::pair<std::string, **int**> > lines\_;
26. **int** lines\_id\_;
27. **int** datum\_channels\_;
28. **int** datum\_height\_;
29. **int** datum\_width\_;
30. **int** datum\_size\_;
31. Blob<Dtype> prefetch\_data\_;
32. Blob<Dtype> prefetch\_label\_;
33. Blob<Dtype> data\_mean\_;
34. Caffe::Phase phase\_;
35. };

修改caffe.proto,在适当的位置添加下面信息，也是参考image\_data写的。

MY\_IMAGE\_DATA = 36;

optional MyImageDataParameter my\_image\_data\_param = 36;

// Message that stores parameters used by MyImageDataLayer  
message MyImageDataParameter {  
  // Specify the data source.  
  optional string source = 1;  
  // For data pre-processing, we can do simple scaling and subtracting the  
  // data mean, if provided. Note that the mean subtraction is always carried  
  // out before scaling.  
  optional float scale = 2 [default = 1];  
  optional string mean\_file = 3;  
  // Specify the batch size.  
  optional uint32 batch\_size = 4;  
  // Specify if we would like to randomly crop an image.  
  optional uint32 crop\_size = 5 [default = 0];  
  // Specify if we want to randomly mirror data.  
  optional bool mirror = 6 [default = false];  
  // The rand\_skip variable is for the data layer to skip a few data points  
  // to avoid all asynchronous sgd clients to start at the same point. The skip  
  // point would be set as rand\_skip \* rand(0,1). Note that rand\_skip should not  
  // be larger than the number of keys in the leveldb.  
  optional uint32 rand\_skip = 7 [default = 0];  
  // Whether or not ImageLayer should shuffle the list of files at every epoch.  
  optional bool shuffle = 8 [default = false];  
  // It will also resize images if new\_height or new\_width are not zero.  
  optional uint32 new\_height = 9 [default = 0];  
  optional uint32 new\_width = 10 [default = 0];  
}

以上每行位置不在一起，可以参考读一个image\_data对应的位置。

[caffe神经网络框架的辅助工具（将图片转换为leveldb格式）](http://blog.csdn.net/lingerlanlan/article/details/31361299)

caffe中负责整个网络输入的datalayer是从leveldb里读取数据的，是一个google实现的非常高效的kv数据库。

因此我们训练网络必须先把数据转成leveldb的格式。

这里我实现的是把一个文件夹的所有图片转成leveldb的格式。

工具使用命令格格式：convert\_imagedata src\_dir dst\_dir attach\_dir channel width height

样例：./convert\_imagedata.bin /home/linger/imdata/collar\_train/ /home/linger/linger/testfile/crop\_train\_db/ /home/linger/linger/testfile/crop\_train\_attachment/ 3 50 50

源代码：

**[cpp]** [view plaincopy](http://blog.csdn.net/lingerlanlan/article/details/31361299)[在CODE上查看代码片](https://code.csdn.net/snippets/407483)

1. #include <google/protobuf/text\_format.h>
2. #include <glog/logging.h>
3. #include <leveldb/db.h>
5. #include <stdint.h>
6. #include <fstream>  // NOLINT(readability/streams)
7. #include <string>
8. #include <set>
9. #include <stdio.h>
10. #include <string.h>
11. #include <stdlib.h>
12. #include <dirent.h>
13. #include <sys/stat.h>
14. #include <unistd.h>
15. #include <sys/types.h>
16. #include "caffe/proto/caffe.pb.h"
17. #include <opencv2/highgui/highgui.hpp>
18. #include <opencv2/highgui/highgui\_c.h>
19. #include <opencv2/imgproc/imgproc.hpp>
21. **using** std::string;
22. **using** **namespace** std;

25. set<string> all\_class\_name;
26. map<string,**int**> class2id;

29. /\*\*
30. \* path:目录
31. \* files：用于保存文件名的vector
32. \* r：是否需要遍历子目录
33. \* return:文件名，不包含路径
34. \*/
35. **void** list\_dir(**const** **char** \*path,vector<string>& files,**bool** r = **false**)
36. {
37. DIR \*pDir;
38. **struct** dirent \*ent;
39. **char** childpath[512];
40. pDir = opendir(path);
41. memset(childpath, 0, **sizeof**(childpath));
42. **while** ((ent = readdir(pDir)) != NULL)
43. {
44. **if** (ent->d\_type & DT\_DIR)
45. {
47. **if** (strcmp(ent->d\_name, ".") == 0 || strcmp(ent->d\_name, "..") == 0)
48. {
49. **continue**;
50. }
51. **if**(r) //如果需要遍历子目录
52. {
53. sprintf(childpath, "%s/%s", path, ent->d\_name);
54. list\_dir(childpath,files);
55. }
56. }
57. **else**
58. {
59. files.push\_back(ent->d\_name);
60. }
61. }
62. sort(files.begin(),files.end());//排序
64. }
66. string get\_classname(string path)
67. {
68. **int** index = path.find\_last\_of('\_');
69. **return** path.substr(0, index);
70. }

73. **int** get\_labelid(string fileName)
74. {
75. string class\_name\_tmp = get\_classname(fileName);
76. all\_class\_name.insert(class\_name\_tmp);
77. map<string,**int**>::iterator name\_iter\_tmp = class2id.find(class\_name\_tmp);
78. **if** (name\_iter\_tmp == class2id.end())
79. {
80. **int** id = class2id.size();
81. class2id.insert(name\_iter\_tmp, std::make\_pair(class\_name\_tmp, id));
82. **return** id;
83. }
84. **else**
85. {
86. **return** name\_iter\_tmp->second;
87. }
88. }
90. **void** loadimg(string path,**char**\* buffer)
91. {
92. cv::Mat img = cv::imread(path, CV\_LOAD\_IMAGE\_COLOR);
93. string val;
94. **int** rows = img.rows;
95. **int** cols = img.cols;
96. **int** pos=0;
97. **for** (**int** c = 0; c < 3; c++)
98. {
99. **for** (**int** row = 0; row < rows; row++)
100. {
101. **for** (**int** col = 0; col < cols; col++)
102. {
103. buffer[pos++]=img.at<cv::Vec3b>(row,col)[c];
104. }
105. }
106. }
108. }
109. **void** convert(string imgdir,string outputdb,string attachdir,**int** channel,**int** width,**int** height)
110. {
111. leveldb::DB\* db;
112. leveldb::Options options;
113. options.create\_if\_missing = **true**;
114. options.error\_if\_exists = **true**;
115. caffe::Datum datum;
116. datum.set\_channels(channel);
117. datum.set\_height(height);
118. datum.set\_width(width);
119. **int** image\_size = channel\*width\*height;
120. **char** buffer[image\_size];
122. string value;
123. CHECK(leveldb::DB::Open(options, outputdb, &db).ok());
124. vector<string> filenames;
125. list\_dir(imgdir.c\_str(),filenames);
126. string img\_log = attachdir+"image\_filename";
127. ofstream writefile(img\_log.c\_str());
128. **for**(**int** i=0;i<filenames.size();i++)
129. {
130. string path= imgdir;
131. path.append(filenames[i]);//算出绝对路径
133. loadimg(path,buffer);
135. **int** labelid = get\_labelid(filenames[i]);
137. datum.add\_label(labelid);
138. datum.set\_data(buffer,image\_size);
139. datum.SerializeToString(&value);
140. snprintf(buffer, image\_size, "%05d", i);
141. printf("\nclassid:%d classname:%s abspath:%s",labelid,get\_classname(filenames[i]).c\_str(),path.c\_str());
142. db->Put(leveldb::WriteOptions(),string(buffer),value);
143. //printf("%d %s\n",i,fileNames[i].c\_str());
145. assert(writefile.is\_open());
146. writefile<<i<<" "<<filenames[i]<<"\n";
148. }
149. **delete** db;
150. writefile.close();
152. img\_log = attachdir+"image\_classname";
153. writefile.open(img\_log.c\_str());
154. set<string>::iterator iter = all\_class\_name.begin();
155. **while**(iter != all\_class\_name.end())
156. {
157. assert(writefile.is\_open());
158. writefile<<(\*iter)<<"\n";
159. //printf("%s\n",(\*iter).c\_str());
160. iter++;
161. }
162. writefile.close();
164. }
166. **int** main(**int** argc, **char**\*\* argv)
167. {
168. **if** (argc < 6)
169. {
170. LOG(ERROR) << "convert\_imagedata src\_dir dst\_dir attach\_dir channel width height";
171. **return** 0;
172. }
173. //./convert\_imagedata.bin  /home/linger/imdata/collarTest/ /home/linger/linger/testfile/dbtest/  /home/linger/linger/testfile/test\_attachment/ 3 250 250
174. //   ./convert\_imagedata.bin /home/linger/imdata/collar\_train/ /home/linger/linger/testfile/crop\_train\_db/ /home/linger/linger/testfile/crop\_train\_attachment/ 3 50 50
175. google::InitGoogleLogging(argv[0]);
176. string src\_dir = argv[1];
177. string src\_dst = argv[2];
178. string attach\_dir = argv[3];
179. **int** channel = atoi(argv[4]);
180. **int** width = atoi(argv[5]);
181. **int** height = atoi(argv[6]);
183. //for test
184. /\*
185. src\_dir = "/home/linger/imdata/collarTest/";
186. src\_dst = "/home/linger/linger/testfile/dbtest/";
187. attach\_dir = "/home/linger/linger/testfile/";
188. channel = 3;
189. width = 250;
190. height = 250;
191. \*/
193. convert(src\_dir,src\_dst,attach\_dir,channel,width,height);


197. }