caffe代码阅读1：blob的实现细节-2016.3.14

caffe代码阅读1：blob的实现细节-2016.3.14

caffe 中 BLOB的实现

一、前言

等着caffe没有膨胀到很大的程度把caffe的代码理一理
（1）第一次阅读Caffe的源码，给人的印象就是里面大量使用了gtest，确实也简化了不少代码，看起来很清晰。
（2）caffe的文档是使用doxygen来生成的，这点在注释里面有体现，对于自己以后的项目也可以借鉴。

二、相关知识：

（1）explicit关键字的作用是禁止隐式转换
比如
A a();
B b = a;// 编译错误
B b(a); //正确
（2）关于const的用法具体参考：
http://blog.csdn.net/Eric_Jo/article/details/4138548

三、具体介绍

BLOB介绍：
看过代码之后，实际上BLOL包含了三类数据
（1）data，前向传播所用到的数据
（2）diff，反向传播所用到的数据
（3）shape，解释data和diff的shape数据
那么围绕这三类数据有对应的方法。
下面给出我的具体的注释：
首先给出blob.h的注释

[cpp] view plain copy

```cpp
#ifndef CAFFE_BLOB_HPP_
#define CAFFE_BLOB_HPP_

#include <algorithm>
#include <string>
#include <vector>

#include "caffe/common.hpp"
#include "caffe/proto/caffe.pb.h"
#include "caffe/syncedmem.hpp"
#include "caffe/util/math_functions.hpp"

const int kMaxBlobAxes = 32;

namespace caffe {

/**
 * @brief A wrapper around SyncedMemory holders serving as the basic
 *        computational unit through which Layer%s, Net%s, and Solver%s
 *        interact.
 *  BLOB是SyncedMemory的包裹器
```

OpenCV（转载）(2)

PRML（转载）(4)

Protocol Buffer（转载）(8)

PS（转载）(2)

Python（转载）(36)

SVM（转载）(13)

Visual Studio debug（转载）(2)

windows问题（转载）(1)

报告（转载）(3)

编程技巧经验（转载）(14)

并行计算（转载）(1)

程序设计小问题(3)

吃（转载）(4)

处理数据（转载）(13)

大牛对话（转载）(12)

大牛列表 传记 故事 （转载）(14)

```
 *
 * TODO(dox): more thorough description.
 */
template <typename Dtype>
class Blob {
 public:
  // 构造函数
  Blob()
       : data_(), diff_(), count_(0), capacity_(0) {}

  /// @brief Deprecated; use <code>Blob(const vector<int>& shape)</code>.
  explicit Blob(const int num, const int channels, const int height,
      const int width);
  explicit Blob(const vector<int>& shape);// 推荐使用这个

  // 成员函数
  /// @brief Deprecated; use <code>Reshape(const vector<int>& shape)</code>.
  void Reshape(const int num, const int channels, const int height,
      const int width);
  /**
   * @brief Change the dimensions of the blob, allocating new memory if
   *        necessary.
   *
   * 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.
   */
  void Reshape(const vector<int>& shape); // 推荐使用这个
  void Reshape(const BlobShape& shape);
  void ReshapeLike(const Blob& other);
  // 输出数据的维度,以空格分隔，最后输出一维维度(total)
```

代码注释

代码注释 (转载)

电脑基础知识 (转载) (12)

调参 (转载) (48)

感悟 (转载) (6)

搞笑 段子 故事 (转载) (9)

机器学习 (转载) (98)

基金 (转载) (5)

健康养生安全 (转载) (8)

降维 (转载) (8)

经济 (转载) (2)

卷积 (转载) (8)

开会 (转载) (5)

科普知识 (转载) (2)

科研经验 (转载) (63)

可视化 (转载) (11)

```
59.   inline string shape_string() const {
60.     ostringstream stream;
61.     for (int i = 0; i < shape_.size(); ++i) {
62.       stream << shape_[i] << " ";
63.     }
64.     stream << "(" << count_ << ")";
65.     return stream.str();
66.   }
67.   inline const vector<int>& shape() const { return shape_; }
68.   /**
69.    * @brief Returns the dimension of the index-th axis (or the negative index-th
70.    *        axis from the end, if index is negative).
71.    *
72.    * @param index the axis index, which may be negative as it will be
73.    *        "canonicalized" using CanonicalAxisIndex.
74.    *        Dies on out of range index.
75.    */
76.    // 计算从给定维度到最后一个维度的
77.   inline int shape(int index) const {
78.     return shape_[CanonicalAxisIndex(index)];
79.   }
80.   // 返回数据的维度
81.   inline int num_axes() const { return shape_.size(); }
82.   // 返回数据的所有维度的相乘,即数据的个数
83.   inline int count() const { return count_; }
84.
85.   /**
86.    * @brief Compute the volume of a slice; i.e., the product of dimensions
87.    *        among a range of axes.
88.    *
89.    * @param start_axis The first axis to include in the slice.
90.    *
91.    * @param end_axis The first axis to exclude from the slice.
92.    */
93.   inline int count(int start_axis, int end_axis) const {
94.     // 判断维度的索引是否在范围内
95.     CHECK_LE(start_axis, end_axis);
```

```
96.     CHECK_GE(start_axis, 0);
97.     CHECK_GE(end_axis, 0);
98.     CHECK_LE(start_axis, num_axes());
99.     CHECK_LE(end_axis, num_axes());
100.    int count = 1;
101.    for (int i = start_axis; i < end_axis; ++i) {
102.      count *= shape(i);
103.    }
104.    return count;
105.  }
106.  /**
107.   * @brief Compute the volume of a slice spanning from a particular first
108.   *        axis to the final axis.
109.   *
110.   * @param start_axis The first axis to include in the slice.
111.   */
112.   // 给定的维度到最后的维度之间包含的数据个数
113.  inline int count(int start_axis) const {
114.    return count(start_axis, num_axes());
115.  }
116.
117.  /**
118.   * @brief Returns the 'canonical' version of a (usually) user-specified axis,
119.   *        allowing for negative indexing (e.g., -1 for the last axis).
120.   *
121.   * @param axis_index the axis index.
122.   *        If 0 <= index < num_axes(), return index.
123.   *        If -num_axes <= index <= -1, return (num_axes() - (-index)),
124.   *        e.g., the last axis index (num_axes() - 1) if index == -1,
125.   *        the second to last if index == -2, etc.
126.   *        Dies on out of range index.
127.   */
128.  // 支持负数维度索引，负数表示从后往前，返回的是正确的维度索引（相当于将负数索引进行的转换）
129.  inline int CanonicalAxisIndex(int axis_index) const {
130.    // 判断是否在范围内[-numaxes, numaxes]
131.    CHECK_GE(axis_index, -num_axes())
132.        << "axis " << axis_index << " out of range for " << num_axes()
```

```
133.         << "-D Blob with shape " << shape_string();
134.     CHECK_LT(axis_index, num_axes())
135.         << "axis " << axis_index << " out of range for " << num_axes()
136.         << "-D Blob with shape " << shape_string();
137.     if (axis_index < 0) {
138.       return axis_index + num_axes();
139.     }
140.     return axis_index;
141.   }
142.
143.   /// @brief Deprecated legacy shape accessor num: use shape(0) instead.
144.   inline int num() const { return LegacyShape(0); }
145.   /// @brief Deprecated legacy shape accessor channels: use shape(1) instead.
146.   inline int channels() const { return LegacyShape(1); }
147.   /// @brief Deprecated legacy shape accessor height: use shape(2) instead.
148.   inline int height() const { return LegacyShape(2); }
149.   /// @brief Deprecated legacy shape accessor width: use shape(3) instead.
150.   inline int width() const { return LegacyShape(3); }
151.   inline int LegacyShape(int index) const {
152.     CHECK_LE(num_axes(), 4)// 检查blob的维度个数是不是小于4，也许以前的blob只有四维，但是现在的blob应
    该为了通用而采用了大于四维的方法
153.         << "Cannot use legacy accessors on Blobs with > 4 axes.";
154.     CHECK_LT(index, 4);// 检查维度索引是不是小于4
155.     CHECK_GE(index, -4);// 检查维度索引是不是大于-4
156.     if (index >= num_axes() || index < -num_axes()) {
157.       // Axis is out of range, but still in [0, 3] (or [-4, -1] for reverse
158.       // indexing) -- this special case simulates the one-padding used to fill
159.       // extraneous axes of legacy blobs.
160.       return 1;
161.     }
162.     return shape(index);
163.   }
164.   // 计算一维线性偏移量
165.   inline int offset(const int n, const int c = 0, const int h = 0,
166.       const int w = 0) const {
167.     CHECK_GE(n, 0);
168.     CHECK_LE(n, num());
```

```
169.     CHECK_GE(channels(), 0);
170.     CHECK_LE(c, channels());
171.     CHECK_GE(height(), 0);
172.     CHECK_LE(h, height());
173.     CHECK_GE(width(), 0);
174.     CHECK_LE(w, width());
175.     return ((n * channels() + c) * height() + h) * width() + w;
176.   }
177.   // 计算一维线性偏移量,只不过参数用的是vector<int>
178.   inline int offset(const vector<int>& indices) const {
179.     CHECK_LE(indices.size(), num_axes());
180.     int offset = 0;
181.     for (int i = 0; i < num_axes(); ++i) {
182.       offset *= shape(i);
183.       if (indices.size() > i) {
184.         CHECK_GE(indices[i], 0);
185.         CHECK_LT(indices[i], shape(i));
186.         offset += indices[i];
187.       }
188.     }
189.     return offset;
190.   }
191.   /**
192.    * @brief Copy from a source Blob.
193.    *
194.    * @param source the Blob to copy from
195.    * @param copy_diff if false, copy the data; if true, copy the diff
196.    * @param reshape if false, require this Blob to be pre-shaped to the shape
197.    *        of other (and die otherwise); if true, Reshape this Blob to other's
198.    *        shape if necessary
199.    * 从给定的blob进行复制，如果copy_diff=true则新的blob复制的是diff,如果reshape=true则改变新blob的形
       状
200.    */
201.   void CopyFrom(const Blob<Dtype>& source, bool copy_diff = false,
202.       bool reshape = false);
203.   // 获取在内存下的数据(前向传播所用的数据)
204.   inline Dtype data_at(const int n, const int c, const int h,
```

```
205.       const int w) const {
206.     return cpu_data()[offset(n, c, h, w)];
207.   }
208.   // 获取在内存下的diff数据(反传数据)
209.   inline Dtype diff_at(const int n, const int c, const int h,
210.       const int w) const {
211.     return cpu_diff()[offset(n, c, h, w)];
212.   }
213.   // 获取在内存下的数据(前向传播所用的数据)
214.   inline Dtype data_at(const vector<int>& index) const {
215.     return cpu_data()[offset(index)];
216.   }
217.   // 获取在内存下的diff数据(反传数据)
218.   inline Dtype diff_at(const vector<int>& index) const {
219.     return cpu_diff()[offset(index)];
220.   }
221.   // 同步内存shared_ptr（不明白share_ptr的可以自行百度，引用计数管理机制）
222.   inline const shared_ptr<SyncedMemory>& data() const {
223.     CHECK(data_);
224.     return data_;
225.   }
226.
227.   inline const shared_ptr<SyncedMemory>& diff() const {
228.     CHECK(diff_);
229.     return diff_;
230.   }
231.
232.   // 属性
233.   const Dtype* cpu_data() const;
234.   void set_cpu_data(Dtype* data);
235.   const int* gpu_shape() const;
236.   const Dtype* gpu_data() const;
237.   const Dtype* cpu_diff() const;
238.   const Dtype* gpu_diff() const;
239.   Dtype* mutable_cpu_data();
240.   Dtype* mutable_gpu_data();
241.   Dtype* mutable_cpu_diff();
```

```
  Dtype* mutable_gpu_diff();
  // 计算
Y=alpha*X+beta*Y

  void Update();
  // 从protobuf序列化文件读取blob对象
  void FromProto(const BlobProto& proto, bool reshape = true);
  // 将对象序列化为protobuf文件
  void ToProto(BlobProto* proto, bool write_diff = false) const;

  /// @brief Compute the sum of absolute values (L1 norm) of the data.
  Dtype asum_data() const;
  /// @brief Compute the sum of absolute values (L1 norm) of the diff.
  Dtype asum_diff() const;
  /// @brief Compute the sum of squares (L2 norm squared) of the data.
  Dtype sumsq_data() const;
  /// @brief Compute the sum of squares (L2 norm squared) of the diff.
  Dtype sumsq_diff() const;

  /// @brief Scale the blob data by a constant factor.
  void scale_data(Dtype scale_factor);
  /// @brief Scale the blob diff by a constant factor.
  void scale_diff(Dtype scale_factor);

  /**
   * @brief Set the data_ shared_ptr to point to the SyncedMemory holding the
   *        data_ of Blob other -- useful in Layer%s which simply perform a copy
   *        in their Forward pass.
   *
   * This deallocates the SyncedMemory holding this Blob's data_, as
   * shared_ptr calls its destructor when reset with the "=" operator.
   */
  void ShareData(const Blob& other);
  /**
   * @brief Set the diff_ shared_ptr to point to the SyncedMemory holding the
   *        diff_ of Blob other -- useful in Layer%s which simply perform a copy
   *        in their Forward pass.
```

```
277.    *
278.    * This deallocates the SyncedMemory holding this Blob's diff_, as
279.    * shared_ptr calls its destructor when reset with the "=" operator.
280.    * 将别的blob的data和响应的diff指针给这个Blob，实现数据的共享。
281.    * 同时需要注意的是这个操作会引起这个Blob里面的SyncedMemory被释放，
282.    * 因为shared_ptr指针被用=重置的时候回调用响应的析构器。
283.    */
284.   void ShareDiff(const Blob& other);
285.   // 判断形状是否相等
286.   bool ShapeEquals(const BlobProto& other);
287.
288.  protected:
289.   // 前向传播的数据
290.   shared_ptr<SyncedMemory> data_;
291.   // diff是反向传播的数据
292.   shared_ptr<SyncedMemory> diff_;
293.   // 旧的形状数据
294.   shared_ptr<SyncedMemory> shape_data_;
295.   // 新的形状数据
296.   vector<int> shape_;
297.   // 数据的个数
298.   int count_;
299.   // 容量
300.   int capacity_;
301.
302.   DISABLE_COPY_AND_ASSIGN(Blob);
303. };  // class Blob
304.
305. }  // namespace caffe
306.
307. #endif  // CAFFE_BLOB_HPP_
```

接下来给出blob所对应的实现
blob.cpp的注释

[cpp] view plain copy

```
#include <climits>
#include <vector>

#include "caffe/blob.hpp"
#include "caffe/common.hpp"
#include "caffe/syncedmem.hpp"
#include "caffe/util/math_functions.hpp"

namespace caffe {

// reshape 的具体实现
// 过时的方法最终是调用的新的reshape方法
template <typename Dtype>
void Blob<Dtype>::Reshape(const int num, const int channels, const int height,
    const int width) {
  vector<int> shape(4);
  shape[0] = num;
  shape[1] = channels;
  shape[2] = height;
  shape[3] = width;
  Reshape(shape);
}

// reshape 的具体实现
template <typename Dtype>
void Blob<Dtype>::Reshape(const vector<int>& shape) {
  CHECK_LE(shape.size(), kMaxBlobAxes); //是否小于规定的最大BLOB的维度(35维)
  count_ = 1;
  shape_.resize(shape.size());// 首先将大小设置为vector<int> shape_; 即新的形状数据的大小
  if (!shape_data_ || shape_data_->size() < shape.size() * sizeof(int)) {
    shape_data_.reset(new SyncedMemory(shape.size() * sizeof(int)));//  shared_ptr<SyncedMemory> shape_data_;
  }
  int* shape_data = static_cast<int*>(shape_data_->mutable_cpu_data());
  for (int i = 0; i < shape.size(); ++i) {
    // 检查形状数据是否合法
```

```
36.     CHECK_GE(shape[i], 0);
37.     CHECK_LE(shape[i], INT_MAX / count_) << "blob size exceeds INT_MAX";
38.     // 计算数据个数
39.     count_ *= shape[i];
40.     // 复制shape到新的和旧的形状数据
41.     shape_[i] = shape[i];
42.     shape_data[i] = shape[i];
43.   }
44.   // 判断是否大于存储的容量
45.   if (count_ > capacity_) {
46.     capacity_ = count_;
47.     // 重新分配内存
48.     data_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
49.     diff_.reset(new SyncedMemory(capacity_ * sizeof(Dtype)));
50.   }
51. }
52. 
53. // 所谓的reshape实际上就仅仅是复制了shape的数据而已
54. // 在调用的时候自动乘以shape的数据就可以得到数据，有点tricky
55. template <typename Dtype>
56. void Blob<Dtype>::Reshape(const BlobShape& shape) {
57.   // 维度是否小于35
58.   CHECK_LE(shape.dim_size(), kMaxBlobAxes);
59.   // 复制形状数据
60.   vector<int> shape_vec(shape.dim_size());
61.   for (int i = 0; i < shape.dim_size(); ++i) {
62.     shape_vec[i] = shape.dim(i);
63.   }
64.   // 调用新的reshape函数
65.   Reshape(shape_vec);
66. }
67. 
68. template <typename Dtype>
69. void Blob<Dtype>::ReshapeLike(const Blob<Dtype>& other) {
70.   Reshape(other.shape());
71. }
72. 
```

```
73. template <typename Dtype>
74. Blob<Dtype>::Blob(const int num, const int channels, const int height,
75.     const int width)
76.   // capacity_ must be initialized before calling Reshape
77.   // 技巧，先初始化容量为0，然后用reshape来分配内存了
78.   : capacity_(0) {
79.   Reshape(num, channels, height, width);
80. }
81. 
82. template <typename Dtype>
83. Blob<Dtype>::Blob(const vector<int>& shape)
84.   // capacity_ must be initialized before calling Reshape
85.   : capacity_(0) {
86.   Reshape(shape);
87. }
88. 
89. template <typename Dtype>
90. const int* Blob<Dtype>::gpu_shape() const {
91.   CHECK(shape_data_);
92.   // shared_ptr<SyncedMemory> shape_data_;
93.   // 因此也分gpu_data和cpu_data
94.   return (const int*)shape_data_->gpu_data();
95. }
96. 
97. template <typename Dtype>
98. const Dtype* Blob<Dtype>::cpu_data() const {
99.   CHECK(data_);
100.   / shared_ptr<SyncedMemory> data_;
101.   return (const Dtype*)data_->cpu_data();
102. }
103. 
104. template <typename Dtype>
105. void Blob<Dtype>::set_cpu_data(Dtype* data) {
106.   CHECK(data);
107.   data_->set_cpu_data(data);
108. }
109. 
```

```
110. template <typename Dtype>
111. const Dtype* Blob<Dtype>::gpu_data() const {
112.   CHECK(data_);
113.   return (const Dtype*)data_->gpu_data();
114. }
115. 
116. template <typename Dtype>
117. const Dtype* Blob<Dtype>::cpu_diff() const {
118.   CHECK(diff_);
119.   return (const Dtype*)diff_->cpu_data();
120. }
121. 
122. template <typename Dtype>
123. const Dtype* Blob<Dtype>::gpu_diff() const {
124.   CHECK(diff_);
125.   return (const Dtype*)diff_->gpu_data();
126. }
127. 
128. template <typename Dtype>
129. Dtype* Blob<Dtype>::mutable_cpu_data() {
130.   CHECK(data_);
131.   return static_cast<Dtype*>(data_->mutable_cpu_data());
132. }
133. 
134. template <typename Dtype>
135. Dtype* Blob<Dtype>::mutable_gpu_data() {
136.   CHECK(data_);
137.   return static_cast<Dtype*>(data_->mutable_gpu_data());
138. }
139. 
140. template <typename Dtype>
141. Dtype* Blob<Dtype>::mutable_cpu_diff() {
142.   CHECK(diff_);
143.   return static_cast<Dtype*>(diff_->mutable_cpu_data());
144. }
145. 
146. template <typename Dtype>
```

```
147. Dtype* Blob<Dtype>::mutable_gpu_diff() {
148.   CHECK(diff_);
149.   return static_cast<Dtype*>(diff_->mutable_gpu_data());
150. }
151.
152. // 将其他blob的数据复制到当前的blob中去
153. template <typename Dtype>
154. void Blob<Dtype>::ShareData(const Blob& other) {
155.   CHECK_EQ(count_, other.count());
156.   data_ = other.data();
157. }
158. // 将其他blob的diff数据复制到当前的blob中去
159. template <typename Dtype>
160. void Blob<Dtype>::ShareDiff(const Blob& other) {
161.   CHECK_EQ(count_, other.count());
162.   diff_ = other.diff();
163. }
164.
165. // The "update" method is used for parameter blobs in a Net, which are stored
166. // as Blob<float> or Blob<double> -- hence we do not define it for
167. // Blob<int> or Blob<unsigned int>.
168. template <> void Blob<unsigned int>::Update() { NOT_IMPLEMENTED; }
169. template <> void Blob<int>::Update() { NOT_IMPLEMENTED; }
170.
171.
172. // Update是计算data=-1 * diff + data
173. template <typename Dtype>
174. void Blob<Dtype>::Update() {
175.   // We will perform update based on where the data is located.
176.   switch (data_->head()) {
177.   case SyncedMemory::HEAD_AT_CPU:
178.     // perform computation on CPU
179.     // axpby即alpha * x plus beta *y 这个含义,blas的函数命名真是见名知意
180.     // template <> void caffe_axpy<float>
(const int N, const float alpha, const float* X, float* Y) { cblas_saxpy(N, alpha, X, 1, Y,
 1); }
181.     // caffe_axpy计算的是Y=alpha * X + Y ，其中alpha=-1了这里
```

```
182.     // 存储的时候用到了mutable_cpu_data，防止其他线程访问
183.     caffe_axpy<Dtype>(count_, Dtype(-1),
184.         static_cast<const Dtype*>(diff_->cpu_data()),
185.         static_cast<Dtype*>(data_->mutable_cpu_data()));
186.     break;
187.   case SyncedMemory::HEAD_AT_GPU:
188.   case SyncedMemory::SYNCED:
189. #ifndef CPU_ONLY
190.     // perform computation on GPU
191.     // Y=alpha * X + Y ，其中alpha=-1了这里
192.     caffe_gpu_axpy<Dtype>(count_, Dtype(-1),
193.         static_cast<const Dtype*>(diff_->gpu_data()),
194.         static_cast<Dtype*>(data_->mutable_gpu_data()));
195. #else
196.     NO_GPU;
197. #endif
198.     break;
199.   default:
200.     LOG(FATAL) << "Syncedmem not initialized.";
201.   }
202. }
203. 
204. template <> unsigned int Blob<unsigned int>::asum_data() const {
205.   NOT_IMPLEMENTED;
206.   return 0;
207. }
208. 
209. template <> int Blob<int>::asum_data() const {
210.   NOT_IMPLEMENTED;
211.   return 0;
212. }
213. // 计算data的L1范数
214. template <typename Dtype>
215. Dtype Blob<Dtype>::asum_data() const {
216.   if (!data_) { return 0; }
217.   switch (data_->head()) {
218.   case SyncedMemory::HEAD_AT_CPU:
```

```
219.     return caffe_cpu_asum(count_, cpu_data());
220.   case SyncedMemory::HEAD_AT_GPU:
221.   case SyncedMemory::SYNCED:
222. #ifndef CPU_ONLY
223.   {
224.     Dtype asum;
225.     caffe_gpu_asum(count_, gpu_data(), &asum);
226.     return asum;
227.   }
228. #else
229.     NO_GPU;
230. #endif
231.   case SyncedMemory::UNINITIALIZED:
232.     return 0;
233.   default:
234.     LOG(FATAL) << "Unknown SyncedMemory head state: " << data_->head();
235.   }
236.   return 0;
237. }
238. 
239. template <> unsigned int Blob<unsigned int>::asum_diff() const {
240.   NOT_IMPLEMENTED;
241.   return 0;
242. }
243. 
244. template <> int Blob<int>::asum_diff() const {
245.   NOT_IMPLEMENTED;
246.   return 0;
247. }
248. 
249. // 计算diff的L1范数
250. template <typename Dtype>
251. Dtype Blob<Dtype>::asum_diff() const {
252.   if (!diff_) { return 0; }
253.   switch (diff_->head()) {
254.   case SyncedMemory::HEAD_AT_CPU:
255.     return caffe_cpu_asum(count_, cpu_diff());
```

```
256.   case SyncedMemory::HEAD_AT_GPU:
257.   case SyncedMemory::SYNCED:
258. #ifndef CPU_ONLY
259.   {
260.     Dtype asum;
261.     caffe_gpu_asum(count_, gpu_diff(), &asum);
262.     return asum;
263.   }
264. #else
265.     NO_GPU;
266. #endif
267.   case SyncedMemory::UNINITIALIZED:
268.     return 0;
269.   default:
270.     LOG(FATAL) << "Unknown SyncedMemory head state: " << diff_->head();
271.   }
272.   return 0;
273. }
274.
275. template <> unsigned int Blob<unsigned int>::sumsq_data() const {
276.   NOT_IMPLEMENTED;
277.   return 0;
278. }
279.
280. template <> int Blob<int>::sumsq_data() const {
281.   NOT_IMPLEMENTED;
282.   return 0;
283. }
284.
285. // 计算sum of square of data(L2范数)
286. template <typename Dtype>
287. Dtype Blob<Dtype>::sumsq_data() const {
288.   Dtype sumsq;
289.   const Dtype* data;
290.   if (!data_) { return 0; }
291.   switch (data_->head()) {
292.   case SyncedMemory::HEAD_AT_CPU:
```

```
293.     data = cpu_data();
294.     sumsq = caffe_cpu_dot(count_, data, data);
295.     break;
296.   case SyncedMemory::HEAD_AT_GPU:
297.   case SyncedMemory::SYNCED:
298. #ifndef CPU_ONLY
299.     data = gpu_data();
300.     caffe_gpu_dot(count_, data, data, &sumsq);
301. #else
302.     NO_GPU;
303. #endif
304.     break;
305.   case SyncedMemory::UNINITIALIZED:
306.     return 0;
307.   default:
308.     LOG(FATAL) << "Unknown SyncedMemory head state: " << data_->head();
309.   }
310.   return sumsq;
311. }
312.
313. template <> unsigned int Blob<unsigned int>::sumsq_diff() const {
314.   NOT_IMPLEMENTED;
315.   return 0;
316. }
317.
318. template <> int Blob<int>::sumsq_diff() const {
319.   NOT_IMPLEMENTED;
320.   return 0;
321. }
322.
323. // sum of square of diff
324. template <typename Dtype>
325. Dtype Blob<Dtype>::sumsq_diff() const {
326.   Dtype sumsq;
327.   const Dtype* diff;
328.   if (!diff_) { return 0; }
329.   switch (diff_->head()) {
```

```
330.   case SyncedMemory::HEAD_AT_CPU:
331.     diff = cpu_diff();
332.     sumsq = caffe_cpu_dot(count_, diff, diff);
333.     break;
334.   case SyncedMemory::HEAD_AT_GPU:
335.   case SyncedMemory::SYNCED:
336. #ifndef CPU_ONLY
337.     diff = gpu_diff();
338.     caffe_gpu_dot(count_, diff, diff, &sumsq);
339.     break;
340. #else
341.     NO_GPU;
342. #endif
343.   case SyncedMemory::UNINITIALIZED:
344.     return 0;
345.   default:
346.     LOG(FATAL) << "Unknown SyncedMemory head state: " << data_->head();
347.   }
348.   return sumsq;
349. }
350. 
351. template <> void Blob<unsigned int>::scale_data(unsigned int scale_factor) {
352.   NOT_IMPLEMENTED;
353. }
354. 
355. template <> void Blob<int>::scale_data(int scale_factor) {
356.   NOT_IMPLEMENTED;
357. }
358. 
359. // 将data部分乘以一个因子scale_factor
360. template <typename Dtype>
361. void Blob<Dtype>::scale_data(Dtype scale_factor) {
362.   Dtype* data;
363.   if (!data_) { return; }
364.   switch (data_->head()) {
365.   case SyncedMemory::HEAD_AT_CPU:
366.     data = mutable_cpu_data();
```

```
367.     caffe_scal(count_, scale_factor, data);
368.     return;
369.   case SyncedMemory::HEAD_AT_GPU:
370.   case SyncedMemory::SYNCED:
371. #ifndef CPU_ONLY
372.     data = mutable_gpu_data();
373.     caffe_gpu_scal(count_, scale_factor, data);
374.     return;
375. #else
376.     NO_GPU;
377. #endif
378.   case SyncedMemory::UNINITIALIZED:
379.     return;
380.   default:
381.     LOG(FATAL) << "Unknown SyncedMemory head state: " << data_->head();
382.   }
383. }
384. 
385. template <> void Blob<unsigned int>::scale_diff(unsigned int scale_factor) {
386.   NOT_IMPLEMENTED;
387. }
388. 
389. template <> void Blob<int>::scale_diff(int scale_factor) {
390.   NOT_IMPLEMENTED;
391. }
392. // 将diff部分乘以一个因子sacle_factor
393. template <typename Dtype>
394. void Blob<Dtype>::scale_diff(Dtype scale_factor) {
395.   Dtype* diff;
396.   if (!diff_) { return; }
397.   switch (diff_->head()) {
398.   case SyncedMemory::HEAD_AT_CPU:
399.     diff = mutable_cpu_diff();
400.     caffe_scal(count_, scale_factor, diff);
401.     return;
402.   case SyncedMemory::HEAD_AT_GPU:
403.   case SyncedMemory::SYNCED:
```

```
#ifndef CPU_ONLY
    diff = mutable_gpu_diff();
    caffe_gpu_scal(count_, scale_factor, diff);
    return;
#else
    NO_GPU;
#endif
  case SyncedMemory::UNINITIALIZED:
    return;
  default:
    LOG(FATAL) << "Unknown SyncedMemory head state: " << diff_->head();
  }
}

// 两个blob是否shape一样
template <typename Dtype>
bool Blob<Dtype>::ShapeEquals(const BlobProto& other) {
  // 判断是否是旧的blob
  if (other.has_num() || other.has_channels() ||
      other.has_height() || other.has_width()) {
    // Using deprecated 4D Blob dimensions --
    // shape is (num, channels, height, width).
    // Note: we do not use the normal Blob::num(), Blob::channels(), etc.
    // methods as these index from the beginning of the blob shape, where legacy
    // parameter blobs were indexed from the end of the blob shape (e.g., bias
    // Blob shape (1 x 1 x 1 x N), IP layer weight Blob shape (1 x 1 x M x N)).
    return shape_.size() <= 4 &&
           LegacyShape(-4) == other.num() &&
           LegacyShape(-3) == other.channels() &&
           LegacyShape(-2) == other.height() &&
           LegacyShape(-1) == other.width();
  }
  // 如果不是旧的blob则直接判断
  vector<int> other_shape(other.shape().dim_size());
  for (int i = 0; i < other.shape().dim_size(); ++i) {
    other_shape[i] = other.shape().dim(i);
  }
```

```
441.   return shape_ == other_shape;
442. }
443.
444. // 从别的blob进行复制
445. template <typename Dtype>
446. void Blob<Dtype>::CopyFrom(const Blob& source, bool copy_diff, bool reshape) {
447.   if (source.count() != count_ || source.shape() != shape_) {
448.     if (reshape) {
449.       ReshapeLike(source);// 复制shape数据
450.     } else {
451.       LOG(FATAL) << "Trying to copy blobs of different sizes.";
452.     }
453.   }
454.   switch (Caffe::mode()) {
455.   case Caffe::GPU:
456.     // GPU复制diff
457.     if (copy_diff) {
458.         // 这都用 template <> void caffe_copy<float>
(const int N, const float* X, float* Y) { cblas_scopy(N, X, 1, Y, 1); }
459.         // 干嘛要用BLAS里面的运算来复制，真是多余...
460.       caffe_copy(count_, source.gpu_diff(),
461.           static_cast<Dtype*>(diff_->mutable_gpu_data()));
462.     } else {
463.       caffe_copy(count_, source.gpu_data(),
464.           static_cast<Dtype*>(data_->mutable_gpu_data()));
465.     }
466.     break;
467.   case Caffe::CPU:
468.     // CPU复制diff
469.     if (copy_diff) {
470.       caffe_copy(count_, source.cpu_diff(),
471.           static_cast<Dtype*>(diff_->mutable_cpu_data()));
472.     } else {
473.       caffe_copy(count_, source.cpu_data(),
474.           static_cast<Dtype*>(data_->mutable_cpu_data()));
475.     }
476.     break;
```

```
477.   default:
478.     LOG(FATAL) << "Unknown caffe mode.";
479.   }
480. }
481.
482. template <typename Dtype>
483. void Blob<Dtype>::FromProto(const BlobProto& proto, bool reshape) {
484.   // copy shape
485.   if (reshape) {
486.     vector<int> shape;
487.     if (proto.has_num() || proto.has_channels() ||
488.         proto.has_height() || proto.has_width()) {
489.       // Using deprecated 4D Blob dimensions --
490.       // shape is (num, channels, height, width).
491.       // 如果是旧的blob直接转换为新的blob中的shape数据
492.       shape.resize(4);
493.       shape[0] = proto.num();
494.       shape[1] = proto.channels();
495.       shape[2] = proto.height();
496.       shape[3] = proto.width();
497.     } else {
498.       shape.resize(proto.shape().dim_size());
499.       for (int i = 0; i < proto.shape().dim_size(); ++i) {
500.         shape[i] = proto.shape().dim(i);
501.       }
502.     }
503.     Reshape(shape);// 复制shape数据到当前blob
504.   } else {
505.     CHECK(ShapeEquals(proto)) << "shape mismatch (reshape not set)";
506.   }
507.   // copy data
508.   Dtype* data_vec = mutable_cpu_data();// 获取当前的blob在内存上的数据指针，该指针是互斥的
509.   if (proto.double_data_size() > 0) {// data
510.     CHECK_EQ(count_, proto.double_data_size());
511.     for (int i = 0; i < count_; ++i) {
512.       data_vec[i] = proto.double_data(i);
513.     }
```

```
514.   } else {
515.     CHECK_EQ(count_, proto.data_size());
516.     for (int i = 0; i < count_; ++i) {
517.       data_vec[i] = proto.data(i);
518.     }
519.   }
520.   // copy diff
521.   if (proto.double_diff_size() > 0) {// diff
522.     CHECK_EQ(count_, proto.double_diff_size());
523.     Dtype* diff_vec = mutable_cpu_diff();// 获取当前的diff在内存上的数据指针，该指针是互斥的
524.     for (int i = 0; i < count_; ++i) {
525.       diff_vec[i] = proto.double_diff(i);
526.     }
527.   } else if (proto.diff_size() > 0) {
528.     CHECK_EQ(count_, proto.diff_size());
529.     Dtype* diff_vec = mutable_cpu_diff();
530.     for (int i = 0; i < count_; ++i) {
531.       diff_vec[i] = proto.diff(i);
532.     }
533.   }
534. }
535. 
536. // BlobProto和BlobShape是protobuf定义的，其中一些函数是自动生成的
537. // mutable_shape、add_dim、clear_double_data、clear_double_diff、add_double_data
538. // add_double_diff等
539. // 见src/caffe/proto/caffe.proto
540. template <>
541. void Blob<double>::ToProto(BlobProto* proto, bool write_diff) const {
542.   proto->clear_shape();
543.   // 存shape
544.   for (int i = 0; i < shape_.size(); ++i) {
545.     proto->mutable_shape()->add_dim(shape_[i]);
546.   }
547. 
548.   proto->clear_double_data();
549.   proto->clear_double_diff();
550.   // 存data
```

```
551.   const double* data_vec = cpu_data();
552.   for (int i = 0; i < count_; ++i) {
553.     proto->add_double_data(data_vec[i]);
554.   }
555.   // 存diff
556.   if (write_diff) {
557.     const double* diff_vec = cpu_diff();
558.     for (int i = 0; i < count_; ++i) {
559.       proto->add_double_diff(diff_vec[i]);
560.     }
561.   }
562. }
563. 
564. template <>
565. void Blob<float>::ToProto(BlobProto* proto, bool write_diff) const {
566.   proto->clear_shape();
567.   for (int i = 0; i < shape_.size(); ++i) {
568.     proto->mutable_shape()->add_dim(shape_[i]);
569.   }
570.   proto->clear_data();
571.   proto->clear_diff();
572.   const float* data_vec = cpu_data();
573.   for (int i = 0; i < count_; ++i) {
574.     proto->add_data(data_vec[i]);
575.   }
576.   if (write_diff) {
577.     const float* diff_vec = cpu_diff();
578.     for (int i = 0; i < count_; ++i) {
579.       proto->add_diff(diff_vec[i]);
580.     }
581.   }
582. }
583. 
584. INSTANTIATE_CLASS(Blob);
585. template class Blob<int>;
586. template class Blob<unsigned int>;
587. 
```

```
588. }  // namespace caffe
```

总结:
还是那句老话，read the fxxx source code.
多翻caffe的issue看

参考：
[1]caffe源码分析另一个，写的也挺好。
http://www.cnblogs.com/louyihang-loves-baiyan/
http://www.cnblogs.com/louyihang-loves-baiyan/p/5149628.html
[2]常用的BLAS含义参考
http://www.cnblogs.com/huashiyiqike/p/3886670.html
http://www.netlib.org/blas/
[3]protobuf的参考
http://***/Article/34963

分类: caffe源码解析（转载）

« 上一篇：Visual Studio的调试技巧
» 下一篇：C++常用的#include头文件总结

最新IT新闻:
· 卡西尼号：不是永别，是远方的游子回家
· 乐视网：贾跃亭未按承诺将减持资金借予公司使用
· 百世物流削减IPO融资规模 阿里巴巴拟投资1亿美元
· HTC难兄难弟：华硕利润创七年新低后也要追求利润优先
· 顺丰将在湖北鄂州建国际物流机场 预计年内开建
» 更多新闻...

最新知识库文章:
· Google 及其云智慧
· 做到这一点，你也可以成为优秀的程序员
· 写给立志做码农的大学生

· 架构腐化之谜
· 学会思考，而不只是编程
» 更多知识库文章...
