Caffe中LMDB的使用

发表于 2015-05-25 | 分类于 academic | 2条评论

最近做实验，要用Caffe提取CNN特征。官方的`extract_feature.bin`很好用，但是输出的特征是放在LMDB里的。以前嫌LMDB麻烦，一直都图方便直接用`ImageDataLayer`来读原始图像。这次绕不过去了，就顺便研究了一下Caffe对LMDB的使用，一些心得写下来和大家分享一下。提取特征的内容下一篇再写。

Caffe中DataLayer默认的数据格式是LMDB。许多example中提供的输入数据是LMDB格式。使用extract_features.bin提取特征时支持的输出格式之一也是LMDB。LMDB在Caffe的IO功能中有相当重要的地位。因此，搞明白如何存取Caffe的LMDB数据，对于我们使用Caffe是很有帮助的。

LMDB

Caffe使用LMDB来存放训练/测试用的数据集，以及使用网络提取出的feature（为了方便，以下还是统称数据集）。数据集的结构很简单，就是大量的矩阵/向量数据平铺开来。数据之间没有什么关联，数据内没有复杂的对象结构，就是向量和矩阵。既然数据并不复杂，Caffe就选择了LMDB这个简单的数据库来存放数据。

LMDB的全称是**Lightning Memory-Mapped Database**，闪电般的内存映射数据库。它文件结构简单，一个文件夹，里面一个数据文件，一个锁文件。数据随意复制，随意传输。它的访问简单，不需要运行单独的数据库管理进程，只要在访问数据的代码里引用LMDB库，访问时给文件路径即可。

图像数据集归根究底从图像文件而来。既然有ImageDataLayer可以直接读取图像文件，为什么还要用数据库来放数据集，增加读写的麻烦呢？我认为，Caffe引入数据库存放数据集，是为了减少IO开销。读取大量小文件的开销是非常大的，尤其是在机械硬盘上。LMDB的整个数据库放在一个文件里，避免了文件系统寻址的开销。LMDB使用内存映射的方式访问文件，使得文件内寻址的开销非常小，使用指针运算就能实现。数据库单文件还能减少数据集复制/传输过程的开销。一个几万，几十万文件的数据集，不管是直接复制，还是打包再解包，过程都无比漫长而痛苦。LMDB数据库只有一个文件，你的介质有多块，就能复制多快，不会因为文件多而慢如蜗牛。

Caffe中的LMDB数据

接下来要介绍Caffe是如何使用LMDB存放数据的。

Caffe中的LMDB数据大约有两类：一类是输入`DataLayer`的训练/测试数据集；另一类则

是 `extract_feature` 输出的特征数据。

Datum数据结构

首先需要注意的是，Caffe并不是把向量和矩阵直接放进数据库的，而是将数据通过caffe.proto里定义的一个 `datum` 类来封装。数据库里放的是一个个的 `datum` 序列化成的字符串。Datum的定义摘录如下：

```
message Datum {
  optional int32 channels = 1;
  optional int32 height = 2;
  optional int32 width = 3;
  // the actual image data, in bytes
  optional bytes data = 4;
  optional int32 label = 5;
  // Optionally, the datum could also hold float data.
  repeated float float_data = 6;
  // If true data contains an encoded image that need to be decoded
  optional bool encoded = 7 [default = false];
}
```

一个Datum有三个维度， `channels` , `height` ,和 `width` ，可以看做是少了num维度的 `Blob` 。存放数据的地方有两个： `byte_data` 和 `float_data` ，分别存放整数型和浮点型数据。图像数据一般是整形，放在 `byte_data` 里，特征向量一般是浮点型，放在 `float_data` 里。 `label` 存放数据的类别标签，是整数型。 `encoded` 标识数据是否需要被解码（里面有可能放的是JPEG或者PNG之类经过编码的数据）。

Datum这个数据结构将数据和标签封装在一起，兼容整形和浮点型数据。经过Protobuf编译后，可以在Python和C++中都提供高效的访问。同时Protubuf还为它提供了序列化与反序列化的功能。存放进LMDB的就是 `Datum` 序列化生成的字符串。

Caffe中读写LMDB的代码

要想知道Caffe是如何使用LMDB的，最好的方法当然是去看Caffe的代码。Caffe中关于LMDB的代码有三类：生成数据集、读取数据集、生成特征向量。接下来就分别针对三者进行分析。

生成数据集

生成数据集的代码在examples，随数据集提供，比如MNIST。

首先，创建访问LMDB所需的一些变量：

```
MDB_env *mdb_env;
MDB_dbi mdb_dbi;
MDB_val mdb_key, mdb_data;
MDB_txn *mdb_txn;
...
```

`mdb_env` 是整个数据库环境的句柄， `mdb_dbi` 是环境中一个数据库的句柄， `mdb_key` 和 `mdb_data` 用来存放向数据库中输入数据的“值”。 `mdb_txn` 是数据库事物操作的句柄，”txn”是”transaction”的缩写。

然后，创建数据库环境，创建并打开数据库：

```
if (db_backend == "lmdb") {  // lmdb
  LOG(INFO) << "Opening lmdb " << db_path;
  CHECK_EQ(mkdir(db_path, 0744), 0)
      << "mkdir " << db_path << "failed";
```

```
 5   CHECK_EQ(mdb_env_create(&mdb_env), MDB_SUCCESS) << "mdb_env_create failed";
 6   CHECK_EQ(mdb_env_set_mapsize(mdb_env, 1099511627776), MDB_SUCCESS)  // 1TB
 7       << "mdb_env_set_mapsize failed";
 8   CHECK_EQ(mdb_env_open(mdb_env, db_path, 0, 0664), MDB_SUCCESS)
 9       << "mdb_env_open failed";
10   CHECK_EQ(mdb_txn_begin(mdb_env, NULL, 0, &mdb_txn), MDB_SUCCESS)
11       << "mdb_txn_begin failed";
12   CHECK_EQ(mdb_open(mdb_txn, NULL, 0, &mdb_dbi), MDB_SUCCESS)
13       << "mdb_open failed. Does the lmdb already exist? ";
14 } else {
15   LOG(FATAL) << "Unknown db backend " << db_backend;
16 }
```

第3行代码为数据库创建文件夹，如果文件夹已经存在，程序会报错退出。也就是说，程序不会覆盖已有的数据库。已有的数据库如果不要了，需要手动删除。第13行处创建并打开了一个数据库。需要注意的是，LMDB的一个环境中是可以有多个数据库的，数据库之间以名字区分。`mdb_open()`的第二个参数实际上就是数据库的名称(`char *`)。当一个环境中只有一个数据库的时候，这个参数可以给`NULL`。

最后，为每一个图像创建`Datum`对象，向对象内写入数据，然后将其序列化成字符串，将字符串放入数据库中：

```
 1 Datum datum;
 2 datum.set_channels(1);
 3 datum.set_height(rows);
 4 datum.set_width(cols);
 5 for (int item_id = 0; item_id < num_items; ++item_id) {
 6   image_file.read(pixels, rows * cols);
 7   label_file.read(&label, 1);
 8   datum.set_data(pixels, rows*cols);
 9   datum.set_label(label);
10   snprintf(key_cstr, kMaxKeyLength, "%08d", item_id);
11   datum.SerializeToString(&value);
12   string keystr(key_cstr);
13
14   // Put in db
15   if (db_backend == "lmdb") {  // lmdb
16     mdb_data.mv_size = value.size();
17     mdb_data.mv_data = reinterpret_cast<void*>(&value[0]);
18     mdb_key.mv_size = keystr.size();
19     mdb_key.mv_data = reinterpret_cast<void*>(&keystr[0]);
20     CHECK_EQ(mdb_put(mdb_txn, mdb_dbi, &mdb_key, &mdb_data, 0), MDB_SUCCESS)
21         << "mdb_put failed";
22   } else {
23     LOG(FATAL) << "Unknown db backend " << db_backend;
24   }
25
26   if (++count % 1000 == 0) {
27     // Commit txn
28     if (db_backend == "lmdb") {  // lmdb
29       CHECK_EQ(mdb_txn_commit(mdb_txn), MDB_SUCCESS)
30           << "mdb_txn_commit failed";
31       CHECK_EQ(mdb_txn_begin(mdb_env, NULL, 0, &mdb_txn), MDB_SUCCESS)
32           << "mdb_txn_begin failed";
33     } else {
34       LOG(FATAL) << "Unknown db backend " << db_backend;
35     }
36   }
37 }
```

放入数据的Key是图像的编号，前面补0至8位。需要注意的是18至21行，`MDB_val`类型的`mdb_data`和`mdb_key`中存放的是数据来源的指针，以及数据的长度。第20行的`mdb_put()`函数将数据存入数据库。每隔1000个图像commit一次数据库。只有commit之后，数据才真正写入磁盘。

读取数据集

Caffe中读取LMDB数据集的代码是`DataLayer`，用在网络的最下层，提供数据。`DataLayer`采用顺序遍历的方式读取数据，不支持打乱数据顺序，只能随机跳过前若干个数据。

首先，在`DataLayer`的`DataLayerSetUp`方法中，打开数据库，并获取迭代器`cursor_`：

```
db_.reset(db::GetDB(this->layer_param_.data_param().backend()));
db_->Open(this->layer_param_.data_param().source(), db::READ);
cursor_.reset(db_->NewCursor());
```

然后，在每一次的数据预取时，`InternalThreadEntry()`方法中，从数据库中读取字符串，反序列化为Datum对象，再从Datum对象中取出数据：

```
Datum datum;
datum.ParseFromString(cursor_->value());
```

其中，`cursor_->value()`获取序列化后的字符串。`datum.ParseFromString()`方法对字符串进行反序列化。

最后，要将`cursor_`向前推进：

```
cursor_->Next();
if (!cursor_->valid()) {
  DLOG(INFO) << "Restarting data prefetching from start."
      cursor_->SeekToFirst();
}
```

如果`cursor->valid()`返回false，说明数据库已经遍历到头，这时需要将`cursor_`重置回数据库开头。

不支持样本随机排序应该是`DataLayer`的致命弱点。如果数据库的key能够统一，其实可以通过对key随机枚举的方式实现。

#caffe #lmdb

论文阅读:《Language Models for Image Captioning: The Quirks and What Works》 >

< 用Caffe提取深度特征

3月3日 回复 顶 转发

社交帐号登录: 微信 微博 QQ 人人 更多»

说点什么吧...

发布

RayZ's Blog正在使用多说