该部分所要实现的功能：从一个只包含imageData和imageLabel的LMDB，通过将该数据在指定的训练好的网络结构中进行特征提取，将提取到的特征与相应image的Data和Label同时存入到指定路径下的LMDB中。

该部分主要是对extrract\_features.cpp里的template<typename Dtype>

int feature\_extraction\_pipeline(int argc, char\*\* argv)函数进行修改。也只需要修改这一个文件即可，原来的则个文件是读取相应的网络层feature，但是修改后把网络层feature加入了初始的图片lmdb中去，实现了data和feature放在同一个datum中。

使用前准备：1原始的lmdb train和test的文件;2提取特征需要的test网络文件;3提前训练好的caffemodel文件;

该部分主要做了以下修改

1. 在输入参数的第三个位置插入一个参数，该参数表示的是将要输入的LMDB数据库文件所在的位置。
2. 增加一个对数据库的操作，并定义相应的游标，利用该数据库和游标实现对1中LMDB数据库的读取操作；
3. 对读取到的数据库的当前行数据进行反序列化操作，并将该行数据的内容保存在了定义的Datum对象中。；
4. 对游标的位置进行操作，并在每次内循环的结束进行游标的next操作。

其修改后如下所示：

注:对于输入的命令参数进行详细说明：

一共需要八个参数，按顺序如下所示

例如：

./build/tools/extract\_features

mine/lenet\_original.caffemodel

mine/lenet\_extract.prototxt

mine/mnist\_train\_lmdb

ip1

mine/feature\_train

600

Lmdb

CPU

第一个参数表示训练好的模型文件所在的路径；

第二个参数表示模型训练时所用的网络文件的路径；

第三个参数表示将要读取的LMDB文件夹所在的路径；

第四个参数表示将要提取哪些层的特征，该处可以为多个参数，用逗号进行隔开；

第五个参数表示通过该程序生成的LMDB文件夹的路径，注意该路径下不能包含有该名称的文件夹，否则报错；

第六个参数表示要读取多少个batch（每个batch的batch\_size在网络文件中已经进行了定义，batch \* batch\_size = 第三个参数代表的LMDB中所包含的图像的数量）；

第七个参数表示生成的数据库文件是什么类型的数据库，lmdb，leveldb

第八个参数表示执行程序时是使用CPU还是GPU；

template<typename Dtype>

int feature\_extraction\_pipeline(int argc, char\*\* argv) {

::google::InitGoogleLogging(argv[0]);

const int num\_required\_args = NUM\_REQUIRED\_ARGS;

if (argc < num\_required\_args) {

LOG(ERROR) <<

"This program takes in a trained network and an input data layer, and then"

" extract features of the input data produced by the net.\n"

"Usage: extract\_features pretrained\_net\_param"

" feature\_extraction\_proto\_file image\_LMDB\_file extract\_feature\_blob\_name1[,name2,...]"

" save\_feature\_dataset\_name1[,name2,...] num\_mini\_batches db\_type"

" [CPU/GPU] [DEVICE\_ID=0]\n"

"Note: you can extract multiple features in one pass by specifying"

" multiple feature blob names and dataset names separated by ','."

" The names cannot contain white space characters and the number of blobs"

" and datasets must be equal.";

return 1;

}

int arg\_pos = num\_required\_args;

arg\_pos = num\_required\_args;

if (argc > arg\_pos && strcmp(argv[arg\_pos], "GPU") == 0) {

LOG(ERROR) << "Using GPU";

int device\_id = 0;

if (argc > arg\_pos + 1) {

device\_id = atoi(argv[arg\_pos + 1]);

CHECK\_GE(device\_id, 0);

}

LOG(ERROR) << "Using Device\_id=" << device\_id;

Caffe::SetDevice(device\_id);

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

}

else {

LOG(ERROR) << "Using CPU";

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

}

arg\_pos = 0; // the name of the executable

std::string pretrained\_binary\_proto(argv[++arg\_pos]);

// Expected prototxt contains at least one data layer such as

// the layer data\_layer\_name and one feature blob such as the

// fc7 top blob to extract features.

/\*

layers {

name: "data\_layer\_name"

type: DATA

data\_param {

source: "/path/to/your/images/to/extract/feature/images\_leveldb"

mean\_file: "/path/to/your/image\_mean.binaryproto"

batch\_size: 128

crop\_size: 227

mirror: false

}

top: "data\_blob\_name"

top: "label\_blob\_name"

}

layers {

name: "drop7"

type: DROPOUT

dropout\_param {

dropout\_ratio: 0.5

}

bottom: "fc7"

top: "fc7"

}

\*/

std::string feature\_extraction\_proto(argv[++arg\_pos]);

boost::shared\_ptr<Net<Dtype> > feature\_extraction\_net(

new Net<Dtype>(feature\_extraction\_proto, caffe::TEST));

feature\_extraction\_net->CopyTrainedLayersFrom(pretrained\_binary\_proto);

//此处为增加的输入变量，为输入的命令参数的第四个变量

std::string image\_LMDB\_file(argv[++arg\_pos]);

std::string extract\_feature\_blob\_names(argv[++arg\_pos]);

std::vector<std::string> blob\_names;

boost::split(blob\_names, extract\_feature\_blob\_names, boost::is\_any\_of(","));

std::string save\_feature\_dataset\_names(argv[++arg\_pos]);

std::vector<std::string> dataset\_names;

boost::split(dataset\_names, save\_feature\_dataset\_names,

boost::is\_any\_of(","));

CHECK\_EQ(blob\_names.size(), dataset\_names.size()) <<

" the number of blob names and dataset names must be equal";

size\_t num\_features = blob\_names.size();

for (size\_t i = 0; i < num\_features; i++) {

CHECK(feature\_extraction\_net->has\_blob(blob\_names[i]))

<< "Unknown feature blob name " << blob\_names[i]

<< " in the network " << feature\_extraction\_proto;

}

int num\_mini\_batches = atoi(argv[++arg\_pos]);

std::vector<boost::shared\_ptr<db::DB> > feature\_dbs;

std::vector<boost::shared\_ptr<db::Transaction> > txns;

//此处增加对LMDB数据库的读取操作，定义了数据库读取和游标

boost::shared\_ptr<db::DB> imageData\_dbs;

boost::shared\_ptr<db::Cursor> txnsImageData;

imageData\_dbs.reset(db::GetDB("lmdb"));

//打开指定的数据库，并初始化游标

imageData\_dbs->Open(image\_LMDB\_file, db::READ);

txnsImageData.reset(imageData\_dbs->NewCursor());

const char\* db\_type = argv[++arg\_pos];

for (size\_t i = 0; i < num\_features; ++i) {

string ss = dataset\_names[i];

string sss = dataset\_names.at(i);

LOG(INFO) << "Opening dataset " << dataset\_names[i];

boost::shared\_ptr<db::DB> db(db::GetDB(db\_type));

db->Open(dataset\_names.at(i), db::NEW);

feature\_dbs.push\_back(db);

boost::shared\_ptr<db::Transaction> txn(db->NewTransaction());

txns.push\_back(txn);

}

LOG(ERROR) << "Extracting Features";

#ifdef OUTPUT

ofstream outfile\_label("E:/out\_label.txt", ofstream::app); //想指定文件中写入信息

ofstream outfile\_feature\_label("E:/out\_feature\_label.txt", ofstream::app); //想指定文件中写入信息

ofstream outfile\_imageData("E:/out\_imageData.txt", ofstream::app); //想指定文件中写入信息

#endif

Datum datum;

std::vector<int> image\_indices(num\_features, 0);

for (int batch\_index = 0; batch\_index < num\_mini\_batches; ++batch\_index) {

feature\_extraction\_net->Forward();

for (int i = 0; i < num\_features; ++i) {

const boost::shared\_ptr<Blob<Dtype> > feature\_blob =

feature\_extraction\_net->blob\_by\_name(blob\_names[i]);

int batch\_size = feature\_blob->num();

int dim\_features = feature\_blob->count() / batch\_size;

const Dtype\* feature\_blob\_data;

//将此游标指向该数据库的第一条记录

txnsImageData->SeekToFirst();

//此处将数据库的游标指向当前batch的首位置

for (int k = 0; k < batch\_index \* batch\_size; ++k)

{

txnsImageData->Next();

if (!txnsImageData->valid()) {

DLOG(INFO) << "Restarting data prefetching from start.";

txnsImageData->SeekToFirst();

}

}

for (int n = 0; n < batch\_size; ++n)

{

//对datum中的数据进行清零操作

datum.clear\_data();

datum.clear\_float\_data();

datum.clear\_label();

datum.clear\_width();

datum.clear\_height();

//对当前游标指向的行数据txnsImageData->value()进行反序列化操作，并将其数据保存在datum的特定位置处

datum.ParseFromString(txnsImageData->value());//对当前游标指向的数据进行反序列化，并将反序列化之后的数据存到datum结构体中。

#ifdef OUTPUT

outfile\_label << (float)(datum.label()) << " ";

cout << (float)(datum.label()) << " " << endl;

outfile\_imageData << "开始写入第" << batch\_index \* batch\_size + n << "个图像的数据：共有imageData " << datum.width() \* datum.height() << "个。" << endl;

for (int ii = 0; ii < datum.width() \* datum.height();ii++)

{

outfile\_imageData << (int)(datum.data().data()[ii]) << " ";

}

outfile\_imageData << "写入第" << batch\_index \* batch\_size + n << "个图像的数据结束" << endl;

#endif

//将获取到的feture\_blob\_data写入到Datum的float\_data中

#ifdef OUTPUT

outfile\_feature\_label << "开始写入第" << batch\_index \* batch\_size + n << "个图像的数据：共有feature "<< dim\_features<< "维。" << endl;

#endif

feature\_blob\_data = feature\_blob->cpu\_data() +

feature\_blob->offset(n);

for (int s = 0; s < dim\_features; ++s)

{

datum.add\_float\_data((float)(feature\_blob\_data[s]));//该处是将提取到的feature当作当前图片的lebels的值存入datum结构体中。此处的结构体是进行了相应的改进，该Datum结构体中的label是存储的一个float型的数组

#ifdef OUTPUT

outfile\_feature\_label << (float)(datum.float\_data().data()[s])<< " ";

#endif

}

#ifdef OUTPUT

outfile\_feature\_label << "写入第" << batch\_index \* batch\_size + n << "个图像的数据结束" << endl;

cout << "执行第 " << batch\_index \* batch\_size + n << "个图像的操作" << endl;

#endif

string key\_str = caffe::format\_int(image\_indices[i], 10);

string out;

CHECK(datum.SerializeToString(&out));

txns.at(i)->Put(key\_str, out);

++image\_indices[i];

if (image\_indices[i] % 1000 == 0) {

txns.at(i)->Commit();

txns.at(i).reset(feature\_dbs.at(i)->NewTransaction());

LOG(ERROR) << "Extracted features of " << image\_indices[i] <<

" query images for feature blob " << blob\_names[i];

}

//将游标指向下一行记录的首位置

txnsImageData->Next();

} // for (int n = 0; n < batch\_size; ++n)

} // for (int i = 0; i < num\_features; ++i)

} // for (int batch\_index = 0; batch\_index < num\_mini\_batches; ++batch\_index)

// write the last batch

for (int i = 0; i < num\_features; ++i) {

if (image\_indices[i] % 1000 != 0) {

txns.at(i)->Commit();

}

LOG(ERROR) << "Extracted features of " << image\_indices[i] <<

" query images for feature blob " << blob\_names[i];

feature\_dbs.at(i)->Close();

#ifdef OUTPUT

outfile\_label.close();

outfile\_feature\_label.close();

outfile\_imageData.close();

#endif

}