该部分的工作内容如下所述：本部分工作承接上一部分，默认输入的lmdb中保存了label,feature,feature\_size,data的量，并且修改data层使其能够对feature\_map进行训练。

Blob.hpp,Blob.cpp,base\_data\_layer.cpp,base\_data.layer.hpp,data\_layer.cpp,data\_layer.hpp六个层次进行修改，在data的层中增加了两个top的输出，多了“feature”和”feature\_size”这两个东西。

注：在该部分进行调试时需要对caffe项目的主函数进行命令参数的预输入操作：输入的参数包括两个，

例如：./build/tools/caffe train --solver=mine/lenet\_solver.prototxt

第一个参数train表示的是目前执行的是train阶段的操作；

第二个参数表示执行当前的train操作时需要使用到的网络的solver文件的路径

对程序的修改包含以下部分：

1. a、修改**lenet.prototxt**文件内容：在train和test阶段的Data层，将其中的top由原来的两个改为四个，按顺序分别保存的是：：data（存储的是imageData）、label（image的label）、feature\_label（该图像指定层的feature）、feature\_label\_size（该图像指定的层一共获取到了多少个feature）。

b、注意：train和test阶段的Data层所引用的数据库不可为同一个，会导致程序在运行到test部分的时候崩溃，因为程序对train阶段的数据库进行了加锁操作，当test使用同一个数据库时，由于加锁后进程区的互斥性而造成程序在读取数据库时发生错误。

c、对于loss层的修改为：type修改为：EuclideanLoss，bottom改为ip1和featurelabel，top为loss。

d、删除最后一层全连接层，提取倒数第二层的feature,直接接上欧式距离就能达到相应结果。

2、**blob.cpp**和**blob.hpp**文件的修改：该文件中只支持对int型数据的Reshape操作，没有对float型数据的支持，由于需要将Datum中的float\_data数据取出后存入到当前的层，所以需要在这两个文件中添加对float数据的操作。其中，在blob.hpp文件中添加template <typename Dtype>

void Blob<Dtype>::ReshapeFloat(const vector<int>& shape);表示对当前传入数据的float类型进行解析；在blob。Cpp文件中添加该函数的具体实现，其内容为：

template <typename Dtype>

void Blob<Dtype>::ReshapeFloat(const vector<int>& shape) {

CHECK\_LE(shape.size(), kMaxBlobAxes);

count\_ = 1;

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

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

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

}

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

for (int i = 0; i < shape.size(); ++i) {

CHECK\_GE(shape[i], 0);

if (count\_ != 0) {

CHECK\_LE(shape[i], INT\_MAX / count\_) << "blob size exceeds INT\_MAX";

}

count\_ \*= shape[i];

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

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

}

if (count\_ > capacity\_) {

capacity\_ = count\_;

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

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

}

}

3、在**base\_data\_layer.hpp**文件中，在class Batch部分做如下修改：将

public:

Blob<Dtype> data\_, label\_;

修改为：

public:

Blob<Dtype> data\_, label\_, featrue\_label\_, featrue\_label\_size\_;

该处表示：在Batch中可以直接对featrue\_label\_和featrue\_label\_size\_进行操作，原有程序默认只能对data\_和label\_进行操作。

新增bool output\_feature\_;，通过这个来对整个文件是否输入feature控制

1. 修改文件：**data\_layer.hpp：**文件中的virtual inline int MaxTopBlobs() const { return 2; }改为virtual inline int MaxTopBlobs() const { return 4; }，该处表示的是data\_layer层中的top最多能有几个输出。
2. 修改base\_data\_layer.cpp

首先，修改layersetup，使其对output\_feature\_初始化

template <typename Dtype>

void BaseDataLayer<Dtype>::LayerSetUp(const vector<Blob<Dtype>\*>& bottom,

const vector<Blob<Dtype>\*>& top) {

if (top.size() == 1) {

output\_labels\_ = false;

} else {

output\_labels\_ = true;

}

if(top.size() ==4){

output\_feature\_ = true;

}else{

output\_feature\_ = false;

}

data\_transformer\_.reset(

new DataTransformer<Dtype>(transform\_param\_, this->phase\_));

data\_transformer\_->InitRand();

// The subclasses should setup the size of bottom and top

DataLayerSetUp(bottom, top);

}

搜索文件中所有的output\_label\_，并且在之后添加上output\_feature\_的相应值;

在forward\_cpu中，把前向传播的batch向前传送

template <typename Dtype>

void BasePrefetchingDataLayer<Dtype>::Forward\_cpu(

const vector<Blob<Dtype>\*>& bottom, const vector<Blob<Dtype>\*>& top) {

if (prefetch\_current\_) {

prefetch\_free\_.push(prefetch\_current\_);

}

prefetch\_current\_ = prefetch\_full\_.pop("Waiting for data");

// Reshape to loaded data.

top[0]->ReshapeLike(prefetch\_current\_->data\_);

top[0]->set\_cpu\_data(prefetch\_current\_->data\_.mutable\_cpu\_data());

if (this->output\_labels\_) {

// Reshape to loaded labels.

top[1]->ReshapeLike(prefetch\_current\_->label\_);

top[1]->set\_cpu\_data(prefetch\_current\_->label\_.mutable\_cpu\_data());

}

if(this->output\_feature\_){

top[2]->ReshapeLike(prefetch\_current\_->feature\_);

top[2]->set\_cpu\_data(prefetch\_current\_->feature\_.mutable\_cpu\_data());

top[3]->ReshapeLike(prefetch\_current\_->feature\_size\_);

top[3]->set\_cpu\_data(prefetch\_current\_->feature\_size\_.mutable\_cpu\_data());

}

}

6、修改文件：**data\_layer.cpp：**

**a、对函数**DataLayerSetUp进行以下修改：

//修改该函数，主要是对该函数的top返回值进行修改，使得其返回值的数量由2个增加到4个

template <typename Dtype>

void DataLayer<Dtype>::DataLayerSetUp(const vector<Blob<Dtype>\*>& bottom,

const vector<Blob<Dtype>\*>& top) {

//Reshape(bottom,top);

const int batch\_size = this->layer\_param\_.data\_param().batch\_size();

// Read a data point, and use it to initialize the top blob.

Datum& datum = \*(reader\_.full().peek());

datum.data();

// Use data\_transformer to infer the expected blob shape from datum.

//该处主要是对读取到的数据进行imageData部分的数据解析

vector<int> top\_shape = this->data\_transformer\_->InferBlobShape(datum);

this->transformed\_data\_.Reshape(top\_shape);

// Reshape top[0] and prefetch\_data according to the batch\_size.

top\_shape[0] = batch\_size;

top[0]->Reshape(top\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) {

this->prefetch\_[i].data\_.Reshape(top\_shape);

}

LOG(INFO) << "output data size: " << top[0]->num() << ","

<< top[0]->channels() << "," << top[0]->height() << ","

<< top[0]->width();

// label，该部分主要是对读取到的数据进行label部分的数据解析

if (this->output\_labels\_) {

vector<int> label\_shape(1, batch\_size);

top[1]->Reshape(label\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) {

this->prefetch\_[i].label\_.Reshape(label\_shape);

}

if (datum.float\_data\_size() > 0)

{

//该部分主要是对读取到的数据进行float\_data部分的数据解析，该部分存储的是指定层的feature信息

vector<int> feature\_label\_shape(2);

feature\_label\_shape[0] = batch\_size;

feature\_label\_shape[1] = datum.float\_data\_size();

top[2]->ReshapeFloat(feature\_label\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) {

this->prefetch\_[i].featrue\_label\_.ReshapeFloat(feature\_label\_shape);

}

//该部分主要是对读取到的数据进行float\_data\_size部分的数据解析，该部分存储的是指定层的feature信息的维度

vector<int> float\_data\_size\_shape(1, batch\_size);

top[3]->Reshape(float\_data\_size\_shape);

for (int i = 0; i < this->PREFETCH\_COUNT; ++i) {

this->prefetch\_[i].featrue\_label\_size\_.Reshape(float\_data\_size\_shape);

}

}

}

}

b、对函数load\_batch进行的修改：

// This function is called on prefetch thread

//该函数主要是对从数据库中读取出的batch中的数据进行解析，并存入到top中的指定位置，该部分增加了对feature\_label和feature\_label\_size的操作

template<typename Dtype>

void DataLayer<Dtype>::load\_batch(Batch<Dtype>\* batch) {

CPUTimer batch\_timer;

batch\_timer.Start();

double read\_time = 0;

double trans\_time = 0;

CPUTimer timer;

CHECK(batch->data\_.count());

CHECK(this->transformed\_data\_.count());

ofstream outfile1("D:/data\_layer\_out.txt", ios::app);//ios::app表示在原文件末尾追加

if (!outfile1){

cout << "Open the file failure...\n";

exit(0);

}

// Reshape according to the first datum of each batch

// on single input batches allows for inputs of varying dimension.

const int batch\_size = this->layer\_param\_.data\_param().batch\_size();

Datum& datum = \*(reader\_.full().peek());

// Use data\_transformer to infer the expected blob shape from datum.

vector<int> top\_shape = this->data\_transformer\_->InferBlobShape(datum);

this->transformed\_data\_.Reshape(top\_shape);

// Reshape batch according to the batch\_size.

top\_shape[0] = batch\_size;

batch->data\_.Reshape(top\_shape);

Dtype\* top\_data = batch->data\_.mutable\_cpu\_data();

Dtype\* top\_label = NULL;

float\* top\_feature\_label = NULL;//该指针指向feature\_label\_的数据部分

int\* top\_feature\_size\_label = NULL;// 该指针指向feature\_label\_size\_的数据部分

if (this->output\_labels\_) {

top\_label = batch->label\_.mutable\_cpu\_data();

if (datum.float\_data\_size() > 0)

{

top\_feature\_label = (float \*)(batch->featrue\_label\_.mutable\_cpu\_data());

top\_feature\_size\_label = (int \*)(batch->featrue\_label\_size\_.mutable\_cpu\_data());

}

}

for (int item\_id = 0; item\_id < batch\_size; ++item\_id) {

timer.Start();

// get a datum

Datum& datum = \*(reader\_.full().pop("Waiting for data"));

if (datum.float\_data\_size() > 0)

{

//此处定义feature\_label的信息

top\_feature\_size\_label[item\_id] = (int)(datum.float\_data\_size());

//cout << "开始输出batch->featrue\_label\_[" << item\_id << "]：" << endl;

//outfile1 << "开始输出batch->featrue\_label\_[" << item\_id << "]：" << " "<<endl;

for (int feature\_id = 0; feature\_id < top\_feature\_size\_label[item\_id]; ++feature\_id)

{

top\_feature\_label[item\_id \* top\_feature\_size\_label[item\_id] + feature\_id] = (float)(datum.float\_data().data()[feature\_id]);

//outfile1 << item\_id \* top\_feature\_size\_label[item\_id] + feature\_id << " " << (float)(top\_feature\_label[item\_id \* top\_feature\_size\_label[item\_id] + feature\_id]) << " ";

datum.set\_float\_data(feature\_id, 0);

}

//cout << "输出batch->featrue\_label\_[" << item\_id << "]结束！" << endl;

//outfile1 << datum.label() << endl;

//outfile1 << "输出batch->featrue\_label\_[" << item\_id << "]结束！" << " "<<endl;

}

read\_time += timer.MicroSeconds();

timer.Start();

// Apply data transformations (mirror, scale, crop...)

int offset = batch->data\_.offset(item\_id);

this->transformed\_data\_.set\_cpu\_data(top\_data + offset);

this->data\_transformer\_->Transform(datum, &(this->transformed\_data\_));//对此处的datum可以进行修改，以获得其中的各种数据

// Copy label.

if (this->output\_labels\_) {

top\_label[item\_id] = datum.label();//在此处进行修改，将保存在Datum的float\_data中的特征值存到top\_label数组中

}

trans\_time += timer.MicroSeconds();

reader\_.free().push(const\_cast<Datum\*>(&datum));

}

timer.Stop();

batch\_timer.Stop();

DLOG(INFO) << "Prefetch batch: " << batch\_timer.MilliSeconds() << " ms.";

DLOG(INFO) << " Read time: " << read\_time / 1000 << " ms.";

DLOG(INFO) << "Transform time: " << trans\_time / 1000 << " ms.";

outfile1.close();

}