[caffe]的项目架构

2015 年 05 月 07 日　Project Experience

src：源文件

scripts：脚本文件

build：库

src：放 c++生成的.o 文件
lib：放最后生成的 libxxx.a 和 libxxx.so
include：头文件

Makefile 里面

BUILD_DIR：库路径也就是./build
CXX_SRCS：c++源文件
HXX_SRCS：hpp 源文件
CU_SRCS：cu 源文件
CXX_OBJS：对应 c++生成的编译文件
CU_OBJS：对应 cu
OBJ_BUILD_DIR：编译文件路径
OBJS：所有的可链接文件
XXX_BINS：可执行文件路径
CXX_WARNS：警告输出文件
CUDA_LIB_DIR：cuda 库路径
LIBRARIES：库名字，例如 glog gflags protobuf
WARNINGS := -Wall ：编译警告
CXX：g++
COMMON_FLAGS：通用编译选项
NVCCFLAGS += -G：cuda 编译选项
LINKFLAGS：链接选项
LDFLAGS += -L 上面的 LIBRARIES

具体的编译：

-fPIC 作用于编译阶段，告诉编译器产生与位置无关代码(Position-Independent Code)，则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的

-O2 选项告诉 G++产生尽可能小和尽可能快的代码

-shared　此选项将尽量使用动态库，所以生成文件比较小，但是需要系统由动态库。

ar 命令可以用来创建、修改库

ar rcs libxxx.a xx1.o xx2.o，在库中插入模块（替换），ar rcs .build_release/lib/libcaffe.a ，g++ -shared -o .build_release/lib/libcaffe.so
syncedmem.cpp 这个文件是给 cpu 或者 gpu 分配内存

layer_factory.cpp 这个文件 new 了每一个层

ConvolutionParameter（xxxParameter）继承 protobuf 中的 message 类，基本上 caffe 中每一种参数都是这样实现的

总写了一个 layer 类，NeuronLayer 继承它，其他的 xxxlayer 基本上都继承 NeuronLayer

compute_image_mean.cpp 里面有 main 函数，然后编译链接成了可执行文件 compute_image_mean.bin

dump_network.cpp 将模型存成二进制

caffe 自己设计的数据存储类 Blob：4 维存储(num,channels,height,width)，num 相当于 minibatchsize，行主存储，一个 blob 存储了两块数据，一个是原始 data，一个是求导值，并且有两种访问方 const Dtype* cpu_data() const;不能改变数值，Dtype* mutable_cpu_data();能够改变数值，这个 Dtype 是里面存的数据，然后通过 syncedmem 这个类在 cpu 和 gpu 之间同步数据，同步内存

layers，将每一个小的部件都实现出来，基本上都有 Forward_gpu 和 backward，cpu 四种实现，backward 是通过关于输出的 gradient 来计算关于这一层参数的 gradient

net，实现了某种功能和求导，ip，inner_product，fully connected layer。Net::Init()模型初始化

conv_layer.cpp，layerSetUp，Forward_gpu 是用 cblas 实现的

conv_layer.cu，用 cublas 实现,M_是 filter 矩阵的第一维，表示哪一个 channel，K_是输入展开构成 data 矩阵的第一维，N_是输出的空间维度，H 是输入矩阵的第二维，W 是 filter 矩阵的第二维

im2col_gpu 展开输入矩阵，并转化为列主，groups 是指可能某 2 个 input 只跟 4 个 output channel 有关，

1.gflags
文件入口：caffe/tools/caffe.cpp，main 函数位置（对于一个项目而言，第一步就是找到项目入口，即 main 函数位置）

Google gflags 简化命令行参数处理，有-在前面就是命令行 flags，没有-就是命令行参数。传进来的 flags 有不同类型但要自己定义实际意义，例如下面，DEFINE_bool（int32/int64/string…）是一个布尔类型（32 位整型/64 位整形/string 类型…）的 flag，也就是说能传递一个布尔类

型的值进来，这个值代表了 big_menu。然后这个 flag 名字就变成了 FLAGS_big_menu（类型为 bool 型），假如类型为 string 就可以当成正常的 string 类型来用。

DEFINE_bool(big_menu, true, "Include 'advanced' options in the menu listing");
只能在一个文件中定义一次，在自己想要用的文件中 declare，类似于 extern FLAGS_big_menu，一般在一个头文件中 declare，其他文件要用的就直接 include。

DECLARE_bool(big_menu);
RegisterFlagValidator 设置了一个 validator 函数，用来判断命令行传来的参数是否合理

给定义的 flags 赋传进来的参数，在 main 函数一开始，接收了传递的 argc 和 argv，最后一个参数控制 argv，argc 里面的参数是否改变，改变成什么样。

google::ParseCommandLineFlags(&argc, &argv, true);
设置一个 bool 变量变成 false，就是在变量前加上 no，－nobig_menu。

--variable=value　　//非 bool 变量最好这样表示
--variable/--novariable　　//bool 变量最好这样表示
－会停止解析，foo -f1 1 － -f2 2，f1 是 flag，-f2 不是 Z

可以修改 flag 的默认值，要在解析函数之前

```
FLAGS_lib_verbose = true;
ParseCommandLineFlags(...);
```

2.caffe.cpp
宏定义中#func 是 func 的值，##是连接符

SetUsageMessage() 参数是 string 数组，解释一下命令行各参数的意义

GlobalInit()初始化了 google gflags 和 logging，命令行参数已经被解析好了，

假如有输入参数，那么调用宏定义编译好的 GetBrewFunction()来通过传进来的 string 确定应该执行哪个函数，它返回值是函数指针，将函数指针与 string 构成 map，然后通过传递 string 得到调用哪个函数，没有命令行参数就打印一些信息。

针对 train 的过程，solverParameter 是一个类，ReadProtoFromTextFileOrDie 在文件 io.hpp 中，实现在 io.cpp 中，CHECK_NE 判断是否为负数。

google::protobuf::io 是 proto 自己提供的 API 接口，其中 google::protobuf::TextFormat::Parse(input, a);解析文件，其中 input 是 io 的 FileInputStream，a 是通过.proto 生成的类，存放在.pb.h 里面，然后可以直接当成正常类使用。std::shared_ptr 是通过指针保持某个对象的共享拥有权的智能指针。若干个 shared_ptr 对象可以拥有同一个对象；最后一个指向该对象的 shared_ptr 被销毁或重置时，该对象被销毁。然后初始化 solver。

在检查是否有 snapshots，假如有就恢复，或者是否有 weight 的记录，有也恢复。然后调用 solve 函数。

3.google protocol buffers
序列化结构数据，自己定义一次数据如何结构化，protocol 缓存信息是一些有逻辑性信息记录，包括了一系列的名称-值对，可以具体化 optional fields、required fields、repeated fields。既可以从文件中接收，也生成序列化文件.

```
fstream output("myfile", ios::out | ios::binary);
person.SerializeToOstream(&output);
//再读进来
fstream input("myfile", ios::in | ios::binary);
Person person;
person.ParseFromIstream(&input);
```

相对于 xml 优点在于简单、体积小、读取处理时间快、更少产生歧义、更容易产生易于编程的类。protocol 由 text 格式编码成 binary 格式。

自己定义的 protocol 格式文件后缀是 proto（caffe 的在 src/proto 文件夹下），默认是 proto2，如果使用 3 的语法则加上

```
message BlobProto {
    optional BlobShape shape = 7;
    repeated float data = 5 [packed = true];
    repeated float diff = 6 [packed = true];

    //4D dimensions -- deprecated.    Use "shape" instead.
    optional int32 num = 1 [default = 0];
    optional int32 channels = 2 [default = 0];
    optional int32 height = 3 [default = 0];
    optional int32 width = 4 [default = 0];
}
```

首先定义了 message 是 BlobProto，它有 7 个 fields，可以使用 scalar 类型，也可用自己定义的类型，等号后面的数字是一个唯一的编号 tag，它们需要在二进制类型中来分辨不同的 field，不能够变更，0～15 保留使用频繁的 message 元素，因为它只需要 1 个 byte 编码，但是 16～2047 都需要 2 个 byte，最小的 tag 是 1，19000-19999 不能使用，protocol 自己保留。

required：在一个 message 有且只能有一个这种 field
optional：0，1 个
repeated：能重复
[packed=true]能够更有效的编码

[default = 10]末尾加上它，出现在 optional 的 field，要是连这个都没有，默认 string 空、值 0、false、枚举第一个值。

可以在一个.proto 文件中定义多个 message（caffe 里面就是网络结构），注释风格与 c/c++一致。定义好.proto 后执行编译器，对于 c++而言，编译器由.proto（caffe 是在当前目录）生成.h 和.cc，每一个 message 的描述都在文件中。

可以引用其它文件中的 message，也可以在 message 内定义 message

import "myproject/other_protos.proto";
可以新增 message 里面的 type，但是 fields 一定要是 optional 或者 repeated 的.

通过 package caffe;来避免名称冲突，c++的用法是 namespace::bar;在解析时，首先搜索最底层的类名，然后往上。

c++的使用方法：

1.定义自己的.proto
2.使用 protocol buffer 编译器，protoc -I=$SRC_DIR --cpp_out=$DST_DIR $SRC_DIR/xxx.proto，然后生成了 pb.h、pb.cc
3.在代码中使用 protocol buffer API 来读写 messages
编译后自动生成的.h 文件，开始的一部分是生成的类互相调用的方法，然后是生成的类，这些类的方法大致上是一样。的。

5.glog
log，可以记录下命令行的行为、意外中断等等，首先初始化 google 的 logging 库。

google::InitGoogleLogging(argv[0]);
然后在代码中使用如下的形式，一般大写的 LOG 开头的一般都是，可以有条件的 log，例如 LOG_IF 等。

LOG(INFO) << "xxxx " << xxxx << " xxxx";
还有 CHECK 宏指令，假如条件不满足就跳出应用。然后它会将 LOG 生成的提示语句显示到屏幕上。

6.slover.cpp
它有两种构造函数，第一种参数是 SolverParameter（.proto 生成的类），第二种参数是需要解析的文件名，同样也生成了 SolverParameter。然后传入 init，设置随机种子初始化训练、测试网络。传进来的参数可以是已经有的，也可以是文件再解析，由于对于一个网络可能有多种测试，所以可能会初始化多个测试网络。

解析传进来的“prototxt”，生成训练网络，首先初始化 NetParameter 对象，用于放置全部的网络参数， 然后在初始化训练网络的时候，通过 net 变量给出的 proto 文件地址，ReadNetParamsFromTextFileOrDie(param_.net(), &net_param);在初始化的时候 NetParameter 使用的是之前定义好的 v0 参数，此处就按照不同的目标来进行 upgrade 参数。例如调用 UpgradeV0Net()，首先 UpgradeV0PaddingLayers()对输入添加 pad 的大小（也就是设置了一些参数，具体的 pad 过程会在后面代码中实现）。然后调用 UpgradeV0LayerParameter(const

V1LayerParameter& v0_layer_connection, V1LayerParameter* layer_param)针对每一层的 layer，将传进来的第一个参数所有内容都复制给第二个参数，然后再设置 input 和它的维数（此处也是针对每一层来赋值四个维数，但是我不是很清楚怎么得到后面的层的维数？？？）。然后设置 UpgradeNetDataTransformation()设置 DATA、IMAGEDATA、WINDOWDATA。 然后设置 UpgradeV1Net()设置计算方式 argmax、dropout 等。

solve 函数，调用了 restore 来从文件名中解析 solverstate，假如 solverstate 里面有保存的网络文件，那么解析网络文件名，解析出来的网络传递给 net_。然后调用 step 函数，假如是恢复，那么需要恢复上次跑了多少个 epoch（caffe 里面是 iter）、上次的 loss。当到了 test 的迭代次数，则调用 TestAll 函数对每一种 test 进行循环，通过 test_net_id 来判断是哪种 test，test_iter 是为了把所有的测试集都测试完，因为一次迭代只执行 100 个输入，然后调用了 net.cpp 里面的 Forward。

Forward 返回的是这一次迭代的每一层的 loss 之和，然后针对每一种 test，放入不同的 test_score，然后通过 LOG 打印到屏幕上。

然后调用 ApplyUpdate，它首先调用函数 GetLearningRate()来根据不同的 policy 字符串产生新的 base_lr（每次每层的 lr 的变化由它决定），然后调用 ClipGradients，它决定是否需要改变 learning rate。Regularize()计算并累加 weight 的正则项（weight decay）。ComputeUpdateValue()更新参数，有自己的 learning rate，rate 是通过 base_lr（GetLearningRate()）计算得出来的。

Dtype local_rate = rate * net_params_lr[param_id];
然后调用 net.cpp 的 update。

然后保存快照，调用 Snapshot()，WriteProtoToBinaryFile 写入二进制文件。

7.blob.cpp
主要成员是 data，diff（求导），shape。

shape 里面存放了四个变量，num，channel，width，height

7.net.cpp
vector<Blob*> 里面存放的是一系列 blob 的指针，底层可能有多个 blob，例如不同的输入图片 size 等等。

Forward 调用了 ForwardPrefilled，在内调用了 ForwardFromTo，它从给定的层数 id（start）到 end 来调用 Layer 对象的 Forward 函数(它在 layer.hpp 作为 inline 中实现)

8.layer.cpp
它的 Forward 函数，参数分别是两个一系列 blob 指针，bottom、top。调用在父类中是纯虚函数的 reshape，子类中必须实现它，哪个子类调用这个 forward 就使用它实现的 reshape。通过 caffe 的 mode 来选择执行 Forward_cpu 还是 Forward_gpu。Forward_cpu 也是一个纯虚函数，Forward_gpu 是虚函数。执行完网络的运算，然后针对每一个 bottom 来计算它产生

的 loss，通过 caffe_cpu_dot 来实现点乘，底层是用 blas 库实现的。

//#ifndef CPU_ONLY 意思是假如 CPU_ONLY 被定义了，那么就不执行下面的编译。

9.common_layer.hpp
这里面都是实现的一种运算，例如内积、sigmoid、argmax 等等。

10.cudnn
虽然有 hostapi，但是 gpu 使用的数据还是 device 上的。

初始化需要 cudnnCreate()，结束需要 cudnnDestory()来释放资源。可使用多线程多 gpu，使用 cudaSetDevice()分配不同机器使用不同主线程，对于每一个线程它能初 始化一个控制器来使用对应的设备。自动分配到不同设备上去，在初始化与销毁之间保持不变，一个主机线程希望使用不同设别需要 cudnnCreate()

库函数能被主机线程调用，当一个 handle 处理多个线程要小心，不推荐多线程用同一个 handle。

cudnnHandle_t 是指针，掌控了整个 cuDnn 库，handle 必须要传递给每一个函数调用 cudannStatus_t 函数返回状态。