深度学习框架Caffe学习与应用 第2课

本节课内容

- 1. Caffe目录结构
- 2. 深入Caffe源码
- 3. Caffe三级结构
  - Blob源码分析
  - Layer源码分析
  - Net源码分析
- 4. 数据类型——protobuf
  - protobuf是什么？优点？为什么Caffe选择用它？
  - 如何使用protobuf
- 5. 牛刀小试：训练mnist数据集

1. Caffe目录结构

- **data/** 用于存放下载的训练数据
- **docs/** 帮助文档
- **examples/** 代码样例
- **matlab/** MATLAB接口文件
- **python/** PYTHON接口文件
- **models/** 一些配置好的模型参数
- **scripts/** 一些文档和数据会用到的脚本
- **核心代码**
- **tools/** 保存的源码是用于生成二进制处理程序的，caffe在训练时实际是直接调用这些二进制文件
- **include/** Caffe的实现代码的头文件
- **src/** 实现Caffe的源文件

- **src/ 文件结构**
  - **gtest/** google test一个用于测试的库你make runtest时看见的很多绿色RUN OK就是它，这个与caffe的学习无关，不过是个有用的库
  - **caffe/** 关键代码
    - **test/** 用gtest测试caffe的代码
    - **util/** 数据转换时用的一些代码。caffe速度快，很大程度得益于内存设计上的优化（blob数据结构采用proto）和对卷积的优化（部分与im2col相关）
    - **proto/** 即所谓的“Protobuf”，全称“Google Protocol Buffer”，是一种数据存储格式，帮助caffe提速
    - **layers/** 深度神经网络中的基本结构就是一层层互不相同的网络了，这个文件夹下的源文件以及目前位置“src/caffe”中包含所有.cpp文件就是caffe的核心目录下的核心代码了。

2. Caffe核心源码

- **核心代码：**
- **blob[.cpp .h]** 基本的数据结构Blob类
- **common[.cpp .h]** 定义Caffe类
- **internal_thread[.cpp .h]** 使用boost::thread线程库
- **net[.cpp .h]** 网络结构类Net
- **solver[.cpp .h]** 优化方法类Solver
- **data_transformer[.cpp .h]** 输入数据的基本操作类DataTransformer
- **syncedmem[.cpp .h]** 分配内存和释放内存类CaffeMallocHost，用于同步GPU，CPU数据
- **layer[.cpp .h]** 层类Layer
- **layers/** 此文件夹下面的代码全部至少继承了类Layer, 从layer_factory中注册继承

3. Caffe三级结构（Blobs,Layers,Nets）

- **Blob**：用于数据的保存、交换和操作，Caffe基础存储结构
- **Layer**：用于模型和计算的基础
- **Net**：整合连接Layers

- **Blob**
- 在内存中表示4维数组，在caffe/blob.hpp中，维度包括（width_,height_,channels_,num_）
- num_用于存储数据或权值（data）和权值增量（diff）
- **blob.hpp**
- Blob 在caffe源码 blob.hpp中是一个模板类,可以定义任意的数据类型。
- protected 的成员变量有：data_，diff_，shape_，count_，capacity_ ，其中data_ 和 diff_ 是共享SyncedMemory 类（在syncedmem的源码中定义）的智能指针，shape_是int型的vector，count_ 和capacity_ 是整型变量。
- 其成员函数主要有：Reshape 、ReshapeLike、SharedData、 Updata 等等。
- blob.hpp 包含了caffe.pb.h ，说明caffe protobuf 会向blob 传递参数。

- **#include “caffe/proto/caffe.pb.h"**
  - caffe.pb.h是google protocol buffer根据caffe.proto自动生成的，可以到src/caffe/proto/caffe.proto里看下caffe里面用到的各个数据的定义，比如BlobProto，Datum，NetParameter等。使用这个protocol buffer看起来确实方便，一方面可以用文本文件定义结构化的数据类型，另一方面可以生成查询效率更高、占空间更小的二进制文件
- **#include “caffe/common.hpp"**
  - 主要singleton化Caffe类，并封装了boost和CUDA随机数生成的函数，提供了统一的接口。
- **#include “caffe/syncedmem.hpp"**
  - 定义了以下的接口：
    - inline void CaffeMallocHost(void** ptr, size_t size)
    - inline void CaffeFreeHost(void* ptr)
  - 主要是分配内存和释放内存的。而class SyncedMemory定义了内存分配管理和CPU与GPU之间同步的函数。

- **#include “caffe/util/math_functions.hpp"**
- 封装了很多cblas矩阵运算。

caffe_cpu_gemm 函数

caffe_cpu_gemv 函数

caffe_axpy 函数

caffe_set 函数

caffe_add_scalar 函数

caffe_copy 函数

caffe_scal 函数

caffeine_cup_axpby 函数

caffe_add caffe_sub caffe_mul caffe_div 函数

caffe_powx caffe_sqr caffe_exp caffe_abs 函数

int caffe_rng_rand 函数

caffe_nextafer 函数

caffe_cpu_strided_dot 函数

caffe_cpu_hamming_distance 函数

caffe_cpu_asum 函数

caffe_cpu_scale 函数

- **caffe.proto里面BlobProto的定义：**
- 对于BlobProto，可以看到定义了四个optional的int32类型的名字（name）num、channels、height和width，optional意味着Blob可以有一个或者没有这个参数，每个名字（name）后面都有一个数字，这个数字是其名字的一个标签。这个数字就是用来在生成的二进制文件中搜索查询的标签。关于这个数字，1到15会花费1byte的编码空间，16到2047花费2byte。所以一般建议把那些频繁使用的名字的标签设为1到15之间的值。而后面的repeated意味着float类型的data和diff可以重复任意次，而加上[packed = true]是为了更高效的编码。
- 主要数据有两个data和diff，用num、channels、height和width这四个维度来确定数据的具体位置，做一些数据查询和Blob reshape的操作。

```
message BlobProto {
  optional BlobShape shape = 7;
  repeated float data = 5 [packed = true];
  repeated float diff = 6 [packed = true];
  repeated double double_data = 8 [packed = true];
  repeated double double_diff = 9 [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];
}
```

- **Blob主要变量**
  - shared_ptr<SyncedMemory> data_;
  - shared_ptr<SyncedMemory> diff_;
  - shared_ptr<SyncedMemory> shape_data_;
  - vector<int> shape_;
  - int count_;
  - int capacity_;
- BLob只是一个基本的数据结构，因此内部的变量相对较少，首先是data_指针，指针类型是shared_ptr，属于boost库的一个智能指针，这一部分主要用来申请内存存储data，data主要是正向传播的时候用的。同理，diff_主要用来存储偏差，update data，shape_data和shape_都是存储Blob的形状，一个是老版本一个是新版本。count表示Blob中的元素个数，也就是个数*通道数*高度*宽度,capacity表示当前的元素个数，因为Blob可能会reshape。

- **主要函数**
- **1. 构造函数 & 2. reshape函数**
- 构造函数开辟一个内存空间来存储数据，Reshape函数在Layer中的reshape或者forward操作中来adjust dimension。同时在改变Blob大小时，内存将会被重新分配如果内存大小不够了，并且额外的内存将不会被释放。对input的blob进行reshape,如果立马调用Net::Backward是会出错的，因为reshape之后，要么Net::forward或者Net::Reshape就会被调用来将新的input shape 传播到高层。
- **3. count函数**
- 重载很多个count()函数，主要还是为了统计Blob的容量（volume），或者是某一片（slice），从某个axis到具体某个axis的shape乘积（如 “inline int count(int start_axis, int end_axis)” ）。

- **4. data_数据操作函数 & 5. 反向传播导数diff_操作函数**
  - inline Dtype data_at(const int n, const int c, const int h, const int w)
  - inline Dtype diff_at(const int n, const int c, const int h, const int w)
  - inline Dtype data_at(const vector<int>& index)
  - inline Dtype diff_at(const vector<int>& index)
  - inline const shared_ptr<SyncedMemory>& data()
  - inline const shared_ptr<SyncedMemory>& diff()
- 这一部分函数主要通过给定的位置访问数据，根据位置计算与数据起始的偏差offset，在通过cpu_data*指针获得地址

- **6. FromProto/ToProto 数据序列化**
- 将数据序列化，存储到BlobProto，这里说到Proto是谷歌的一个数据序列化的存储格式，可以实现语言、平台无关、可扩展的序列化结构数据格式。
- **7. Update函数**
- 该函数用于参数blob的更新（weight，bias 等减去对应的导数）

8.其他运算函数

- Dtype asum_data() const;//计算data的L1范数(所有元素绝对值之和)
- Dtype asum_diff() const;//计算diff的L1范数
- Dtype sumsq_data() const;//计算data的L2范数(所有元素平方和)
- Dtype sumsq_diff() const;//计算diff的L2范数
- void scale_data(Dtype scale_factor);//将data部分乘以一个因子
- void scale_diff(Dtype scale_factor);//将diff部分乘一个因子

- **Layer**
- 所有的Pooling，Convolve，apply nonlinearities等操作都在这里实现。在Layer中input data用bottom表示output data用top表示。每一层定义了三种操作setup（Layer初始化），forward（正向传导，根据input计算output），backward（反向传导计算，根据output计算input的梯度）。forward和backward有GPU和CPU两个版本的实现。

- 5种衍生Layers:
  - data_layer
  - neuron_layer
  - loss_layer
  - common_layer
  - vision_layer

data_layer

- data_layer主要包含与数据有关的文件。在官方文档中指出data是caffe数据的入口是网络的最低层，并且支持多种格式，在这之中又有5种LayerType：
  - **DATA** 用于LevelDB或LMDB数据格式的输入的类型，输入参数有source, batch_size, (rand_skip), (backend)。后两个是可选。
  - **MEMORY_DATA** 这种类型可以直接从内存读取数据使用时需要调用MemoryDataLayer::Reset，输入参数有batch_size, channels, height, width。
  - **HDF5_DATA** HDF5数据格式输入的类型，输入参数有source, batch_size。
  - **HDF5_OUTPUT** HDF5数据格式输出的类型，输入参数有file_name。
  - **IMAGE_DATA** 图像格式数据输入的类型，输入参数有source, batch_size, (rand_skip), (shuffle), (new_height), (new_width)。
- 其实还有两种WINDOW_DATA, DUMMY_DATA用于测试和预留的接口,不重要。

neuron_layer

- 同样是数据的操作层，neuron_layer实现里大量激活函数，主要是元素级别的操作，具有相同的bottom,top size。
- Caffe中实现了大量激活函数GPU和CPU的都有很多。它们的父类都是NeuronLayer
  - template <typename Dtype>
  - class NeuronLayer : public Layer<Dtype>
- 一般的参数设置格式如下（以ReLU为例）：
- layers {
- name: "relu1"
- type: RELU
- bottom: "conv1"
- top: "conv1"
- }

loss_layer

- Loss层计算网络误差，loss_layer.hpp头文件调用情况：
  - #include "caffe/blob.hpp"
  - #include "caffe/common.hpp"
  - #include "caffe/layer.hpp"
  - #include "caffe/neuron_layers.hpp"
  - #include "caffe/proto/caffe.pb.h"
- 可以看见调用了neuron_layers.hpp，估计是需要调用里面的函数计算Loss，一般来说Loss放在最后一层。caffe实现了大量loss function，它们的父类都是LossLayer。
  - template <typename Dtype>
  - class LossLayer : public Layer<Dtype>

common_layer

- 这一层主要进行的是vision_layer的连接
- 声明了9个类型的common_layer，部分有GPU实现：
  - InnerProductLayer 常常用来作为全连接层
  - SplitLayer 用于一输入对多输出的场合（对blob）
  - FlattenLayer 将n * c * h * w变成向量的格式n *（c * h * w）* 1 * 1
  - ConcatLayer 用于多输入一输出的场合
  - SilenceLayer 用于一输入对多输出的场合（对layer）
  - (Elementwise Operations) 这里面是我们常说的激活函数层Activation Layers。
    - EltwiseLayer
    - SoftmaxLayer
    - ArgMaxLayer
    - MVNLayer

- **vision_layer**
- 主要是实现Convolution和Pooling操作, 主要有以下几个类：
  - **ConvolutionLayer** 最常用的卷积操作
  - **Im2colLayer** 与MATLAB里面的im2col类似，即image-to-column transformation，转换后方便卷积计算
  - **LRNLayer** 全称local response normalization layer，在Hinton论文中有详细介绍ImageNet Classification with Deep Convolutional Neural Networks 。
  - **PoolingLayer** Pooling操作

- **Net**
- Net由一系列的Layer组成(无回路有向图DAG)，Layer之间的连接由一个文本文件描述。模型初始化Net::Init()会产生blob和layer并调用Layer::SetUp。在此过程中Net会报告初始化进程。这里的初始化与设备无关，在初始化之后通过Caffe::set_mode()设置Caffe::mode()来选择运行平台CPU或GPU，结果是相同的。

ProtoBuf

- Caffe中，数据的读取、运算、存储都是采用Google Protocol Buffer来进行的。

- Protocol Buffer(PB)是一种轻便、高效的结构化数据存储格式，可以用于结构化数据串行化，很适合做数据存储或 RPC 数据交换格式。它可用于通讯协议、数据存储等领域的语言无关、平台无关、可扩展的序列化结构数据格式。是一种效率和兼容性都很优秀的二进制数据传输格式，目前提供了C++、Java、Python 三种语言的 API。Caffe采用的是C++和Python的API。

- **protoc --proto_path=IMPORT_PATH --cpp_out=DST_DIR --java_out=DST_DIR --python_out=DST_DIR path/to/file.proto**
- 这里将给出上述命令的参数解释。
- 1. protoc为Protocol Buffer提供的命令行编译工具。
- 2. —proto_path等同于-I选项，主要用于指定待编译的.proto消息定义文件所在的目录，该选项可以被同时指定多个。
- 3. --cpp_out选项表示生成C++代码，--java_out表示生成Java代码，--python_out则表示生成Python代码，其后的目录为生成后的代码所存放的目录。
- 4. path/to/file.proto表示待编译的消息定义文件。
- 注：对于C++而言，通过Protocol Buffer编译工具，可以将每个.proto文件生成出一对.h和.cc的C++代码文件。生成后的文件可以直接加载到应用程序所在的工程项目中。

5. 训练mnist数据集

- 1. 获取数据
  - ./data/mnist/get_mnist.sh
- 2. 将数据转化为lmdb格式
  - ./examples/mnist/create_mnist.sh
- 3. 训练
  - ./examples/mnist/train_lenet.sh
- 注意：
- * 脚本的运行基于$Caffe_Root文件加下的路径执行
- * 训练的时候，如果安装的时候选择了CPU_ONLY的话，在*.prototxt文件中，把“mode:GPU”改成“mode:CPU”

- ./examples/mnist/create_mnist.sh
  - create_mnist.sh是利用caffe-master/build/examples/mnist/的convert_mnist_data.bin工具，将mnist date转化为可用的lmdb格式的文件。并将新生成的2个文件mnist-train-lmdb和 mnist-test-lmdb放于create_mnist.sh同目录下。

- ./examples/mnist/train_lenet.sh
  - 调用工具：./build/tools/caffe train --solver=examples/mnist/lenet_solver.prototxt

```
# The train/test net protocol buffer definition
net: "examples/mnist/lenet_train_test.prototxt" //网络协议具体定义

# test_iter specifies how many forward passes the test should carry out.
# In the case of MNIST, we have test batch size 100 and 100 test iterations,
# covering the full 10,000 testing images.
test_iter: 100 //test迭代次数 如果batch_size =100,则100张图一批，训练100次，则可以覆盖10000张图的需求

# Carry out testing every 500 training iterations.
test_interval: 500 //训练迭代500次，测试一次

# The base learning rate, momentum and the weight decay of the network. //网络参数：学习率，动量，权重的衰减
base_lr: 0.01
momentum: 0.9
weight_decay: 0.0005

# The learning rate policy //学习策略：有固定学习率和每步递减学习率
lr_policy: "inv"
gamma: 0.0001
power: 0.75

# Display every 100 iterations //每迭代100次显示一次
display: 100

# The maximum number of iterations //最大迭代次数
max_iter: 10000

# snapshot intermediate results // 每5000次迭代存储一次数据，路径前缀是<</span>span style="font-family: Arial, Helvetica, sans-
serif;">examples/mnist/lenet</</span>span>
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"

# solver mode: CPU or GPU //使用GPU或者CPU
solver_mode: GPU
```