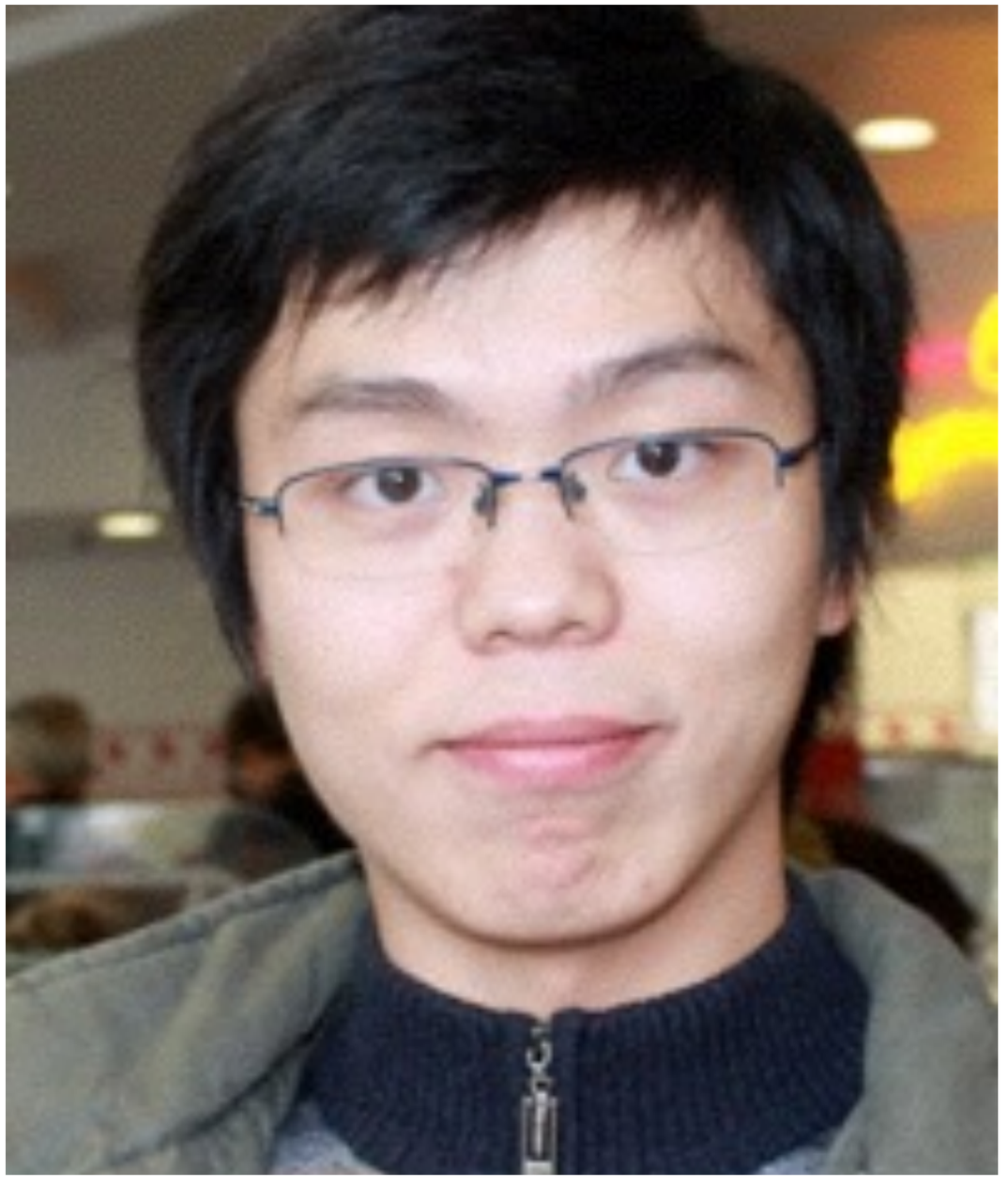
Caffe基础

1. Caffe基础

- 1.1
- Caffe介绍
- 简介Caffe
- Caffe与其他深度学习框架对比
- Caffe的优点和局限性

Caffe介绍

- Caffe全称：Convolutional Architecture for Fast Feature Embedding
- 是一个计算CNN相关算法的框架，用C++和Python实现的
- 作者：Yangqing Jia (贾扬清)
- 加州大学伯克利的Ph.D,现就职于facebook.
- 主页：http://daggerfs.com/



简介Caffe

- 特点
- - 表达方便：模型和优化办法的表达用的是纯文本办法表示，而不是代码
- - 速度快：对于科研来说，我们提供接近于工业化的速度对于大规模数据和当前最牛掰的算法模型是非常重要的
- - 模块化：具有新任务和设置需要的灵活性和扩展性
- - 开放性：科学研究和应用程序可调用同样的代码，参考模型，并且结果可重现。
- - 社区性：学术研究，启动原型和工业应用领域的伙伴以一个BSD-2项目的形式共同讨论和开发 。

Caffe的优点与局限性

- 优点:
- 1. 第一个主流的工业级深度学习工具。
- 2. 专精于图像处理

- 局限性:
- 1. 它有很多扩展，但是由于一些遗留的架构问题，不够灵活且对递归网络和语言建模的支持很差。
- 2. 基于层的网络结构，其扩展性不好，对于新增加的层，需要自己实现（forward, backward and gradient update）

- for req in $(cat requirements.txt); do /home/bjfu/anaconda2/bin/conda install $req; done

- 1.2 Caffe安装

- 操作系统：Ubuntu14.04，解压caffe-master.zip
- 1. 安装环境依赖
- sudo apt-get install libprotobuf-dev libleveldb-dev libsnappy-dev libopencv-dev libhdf5-serial-dev protobuf-compiler cmake
- sudo apt-get install --no-install-recommends libboost-all-dev

- 2. 安装BLAS
- sudo apt-get install libatlas-base-dev
- 3. 安装其他依赖:
- sudo apt-get install libgflags-dev libgoogle-glog-dev liblmdb-dev

- * 安装pycaffe需要的环境:

- 1. 安装python环境：sudo apt-get install python-pip python-dev python

- 2. 安装pycaffe需要的python包：for req in $(cat / path/to/caffe/python/requirements.txt); do pip install $req; done

- 3. 导入系统变量：export PYTHONPATH=/path/to/ caffe/python:$PYTHONPATH

- 4. 使用cmake编译Caffe
- 进入caffe目录：cd ~/caffe
- 创建并进入编译文件夹：mkdir build && cd build
- 执行cmake: cmake -D CPU_ONLY=ON -D CMAKE_INSTALL_PREFIX=/usr/local/ ..
- 执行编译:
- make all
- make install

- 1.3
- Caffe目录结构
- Caffe核心代码

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

.github	Add Github issue template to curb misuse.
cmake	Add Pascal to all cuda architectures
data	Merge pull request #4455 from ShaggO/spaceSupportILSVRC12M
docker	Update Dockerfile to cuDNN v5
docs	docs: include AWS AMI pointer
examples	fix many typos by using codespell
include/caffe	Revert "solver: check and set type to reconcile class and proto"
matlab	fix many typos by using codespell
models	[examples] switch examples + models to Input layers
python	Merge pull request #4343 from nitnelave/python/top_names
scripts	fix many typos by using codespell
src	Add the missing period
tools	fix many typos by using codespell
.Doxyfile	update doxygen config to stop warnings
.gitignore	Ignore Visual Studio Code files.
.travis.yml	Stop setting cache timeout in TravisCI
CMakeLists.txt	[build] (CMake) customisable Caffe version/soversion
CONTRIBUTING.md	[docs] add CONTRIBUTING.md which will appear on GitHub new
CONTRIBUTORS.md	clarify the license and copyright terms of the project
INSTALL.md	installation questions -> caffe-users
LICENSE	copyright 2015
Makefile	Checks inside Xcode for latest OSX SDK (#4840)
Makefile.config.example	[build] note that `make clean` clears build and distribute dirs
README.md	add badge for travis build and license
caffe.cloc	[fix] stop cloc complaint about cu type

- **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。

Caffe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
- **ayer[.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

- 1.4
- Caffe三级结构（Blobs,Layers,Nets）
- Blobs结构
- Layer的五种类型
- Nets结构

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

Blobs结构

- **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。
- 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。

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

#include “caffe/util/math_functions.hpp"

封装了很多cblas矩阵运算。

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：

```
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];
}
```

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。

- 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的五种类型

- **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）l

- ayers {
  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 。

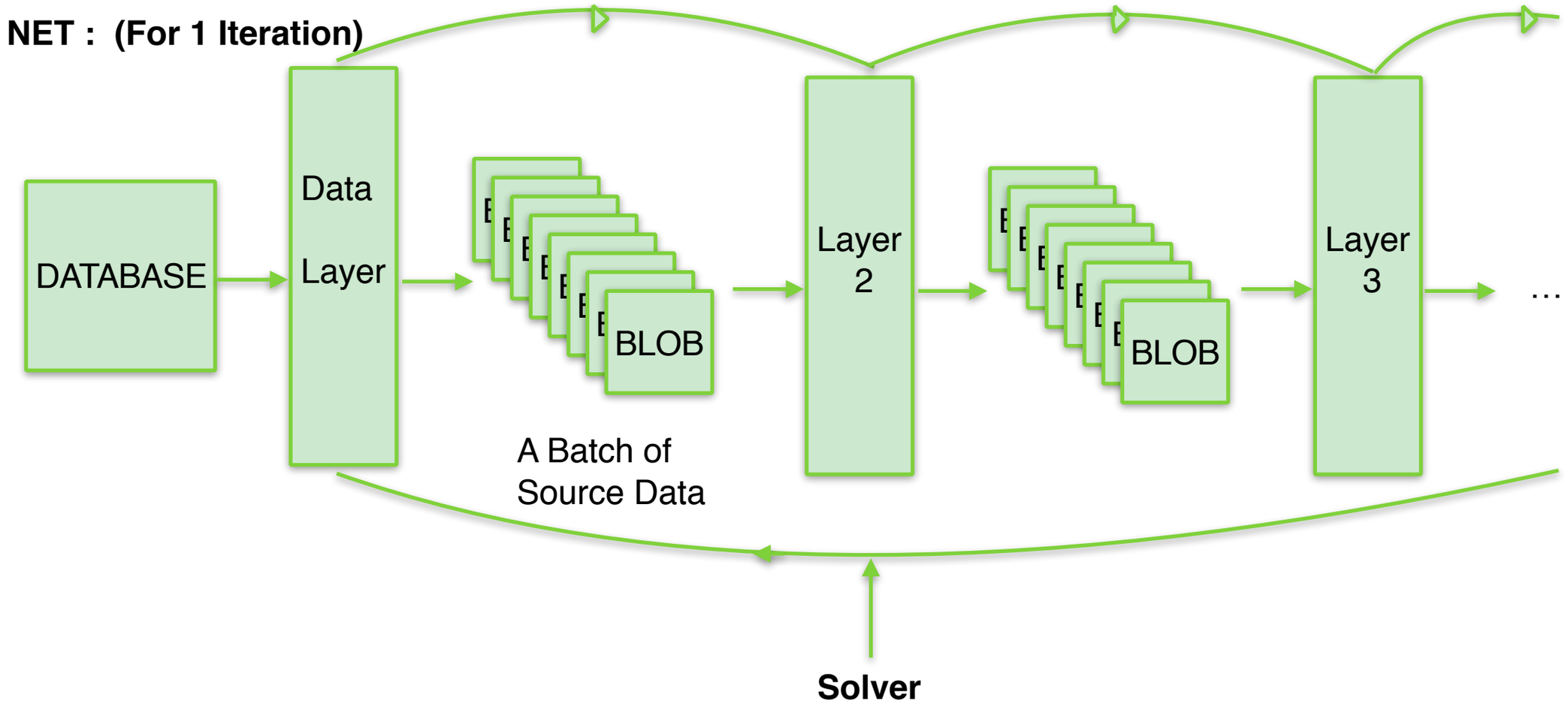
  - **PoolingLayer** Pooling操作

Nets结构

- **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。

- 1.5
- Caffe运行流程
- Protocol Buffer介绍
- 优化求解过程Solver

Caffe运行流程


NET： (For 1 Iteration)
DATABASE
Data Layer
BLOB
Layer 2
BLOB
Layer 3
…
A Batch of Source Data
Solver

Protocol Buffer介绍

- 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。

优化求解过程Solver

- 求解器Solver是什么?
  - Caffe的重中之重（核心）——Solver
  - 负责对模型优化，让损失函数(loss function)达到全局最小。
  - solver的主要作用就是交替调用前向（forward)算法和后向（backward)算法来更新参数，实际上就是一种迭代的优化算法。
- 在每一次的迭代过程中，solver做了这几步工作:
  - 1、调用forward算法来计算最终的输出值，以及对应的loss
  - 2、调用backward算法来计算每层的梯度
  - 3、根据选用的slover方法，利用梯度进行参数更新
  - 4、记录并保存每次迭代的学习率、快照，以及对应的状态。

Solver的重点是最小化损失函数的全局最优问题，对于数据集 D(epoch)，优化目标是在全数据集 D 上损失函数平均值：

$$L(W)=\frac{1}{|D|}\sum_{i}^{|D|}f_w(X^{(i)})+\lambda r(W)$$

其中，$f_w(X^{(i)})$是在数据实例 $X^{(i)}$上的损失函数，$r(W)$为规整项，λ为规整项的权重。数据集 D一般都很大，工程上在每次迭代中使用这个目标函数的随机逼近，即小批量数据 $N<<|D|$个数据实例：

$$L(W)\approx\frac{1}{|N|}\sum_{i}^{|N|}f_w(X^{(i)})+\lambda r(W)$$

模型向前传播计算损失函数 f_w，反向传播计算梯度 ∇f_w。权值增量 ΔW 由求解器通过误差梯度 ∇f_w、规整项梯度$\nabla r(W)$以及其他与方法相关的项求解得到。

- Solver参数配置
  - 查看可配置的参数：https://github.com/BVLC/caffe/blob/master/src/caffe/proto/caffe.proto
  - message SolverParameter {
    …
    }
- 例子：

```
# 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
```

net: "examples/mnist/lenet_train_test.prototxt"

设置深度网络模型。每一个模型就是一个net，需要在一个专门的配置文件中对net进行配置，每个net由许多的layer所组成。注意的是：文件的路径要从caffe的根目录开始，其它的所有配置都是这样。

也可用train_net和test_net来对训练模型和测试模型分别设定：

train_net:"examples/mnist/lenet_train_test.prototxt"

test_net: "examples/mnist/lenet_test_test.prototxt"

test_iter: 100

mnist数据中测试样本总数为10000，一次性执行全部数据效率很低，因此我们将测试数据分成几个批次来执行，每个批次的数量就是batch_size。假设我们设置batch_size为100，则需要迭代100次才能将10000个数据全部执行完。因此test_iter设置为100。执行完一次全部数据，称之为一个epoch。

test_interval: 500

在训练集中每迭代500次，在测试集进行一次测试。

base_lr: 0.01

lr_policy: "inv"

gamma: 0.0001

power: 0.75

这四个参数用于学习率的设置。只要是梯度下降法来求解优化，都会有一个学习率，也叫步长。base_lr用于设置基础学习率，在迭代的过程中，可以对基础学习率进行调整。怎么样进行调整，就是调整的策略，由lr_policy来设置。

lr_policy可以设置为下面这些值，相应的学习率的计算为：

- fixed: 保持base_lr不变.
- step: 如果设置为step,则还需要设置一个stepsize, 返回 base_lr * gamma ^ (floor(iter / stepsize)),其中iter表示当前的迭代次数
- exp: 返回base_lr * gamma ^ iter， iter为当前迭代次数
- inv: 如果设置为inv,还需要设置一个power, 返回base_lr * (1 + gamma * iter) ^ (- power)
- multistep: 如果设置为multistep,则还需要设置一个stepvalue。这个参数和step很相似，step是均匀等间隔变化，而multistep则是根据stepvalue值变化
- poly: 学习率进行多项式误差, 返回 base_lr (1 - iter/max_iter) ^ (power)
- sigmoid: 学习率进行sigmod衰减，返回 base_lr (1/(1 + exp(-gamma * (iter - stepsize))))

weight_decay: 0.0005

momentum： 0.9

type: SGD

— Stochastic Gradient Descent (type: "SGD")

— AdaDelta (type: "AdaDelta")

— Adaptive Gradient (type: "AdaGrad")

— Adam (type: "Adam")

— Nesterov's Accelerated Gradient (type: "Nesterov")

— RMSprop (type: "RMSProp")

2 MNIST手写数据集识别实现

- 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。