

一.数据层及参数

要运行 **caffe**，需要先创建一个模型（**model**），如比较常用的 **Lenet**,**Alex** 等， 而一个模型由多个层（**layer**）构成，每一层又由许多参数组成。所有的参数都定义在 **caffe.proto** 这个文件中。要熟练使用 **caffe**，最重要的就是学会配置文件（**prototxt**）的编写。

层有很多种类型，比如 **Data**,**Convolution**,**Pooling** 等，层之间的数据流动是以 **Blobs** 的方式进行。

今天我们就先介绍一下数据层。

数据层是每个模型的最底层，是模型的入口，不仅提供数据的输入，也提供数据从 **Blobs** 转换成别的格式进行保存输出。通常数据的预处理（如减去均值，放大缩小，裁剪和镜像等），也在这一层设置参数实现。

数据来源可以来自高效的数据库（如 **LevelDB** 和 **LMDB**），也可以直接来自于内存。如果不是很注重效率的话，数据也可来自磁盘的 **hdf5** 文件和图片格式文件。

所有的数据层的都具有的公用参数：先看示例

```
layer {
  name: "cifar"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mean_file: "examples/cifar10/mean.binaryproto"
  }
  data_param {
    source: "examples/cifar10/cifar10_train_lmdb"
    batch_size: 100
    backend: LMDB
  }
}
```

name: 表示该层的名称，可随意取

type: 层类型，如果是 **Data**，表示数据来源于 **LevelDB** 或 **LMDB**。根据数据的来源不同，数据层的类型也不同（后面会详细阐述）。一般在练习的时候，我们都是采用的 **LevelDB** 或 **LMDB** 数据，因此层类型设置为 **Data**。

top 或 bottom: 每一层用 **bottom** 来输入数据，用 **top** 来输出数据。如果只有 **top** 没有 **bottom**，则此层只有输出，没有输入。反之亦然。如果有多个 **top** 或多个 **bottom**，表示有多个 **blobs** 数据的输入和输出。

data 与 label: 在数据层中，至少有一个命名为 **data** 的 **top**。如果有第二个 **top**，一般命名为 **label**。这种(**data**,**label**)配对是分类模型所必需的。

include: 一般训练的时候和测试的时候，模型的层是不一样的。该层（**layer**）是属于训练阶段的层，还是属于测试阶段的层，需要用 **include** 来指定。如果没有 **include** 参数，则表示该层既在训练模型中，又在测试模型中。

Transformations: 数据的预处理，可以将数据变换到定义的范围内。如设置 **scale** 为

0.00390625，实际上就是 $1/255$ ，即将输入数据由 0-255 归一化到 0-1 之间

其它的数据预处理也在这个地方设置：

```
transform_param {
  scale: 0.00390625
  mean_file_size: "examples/cifar10/mean.binaryproto"
  # 用一个配置文件来进行均值操作
  mirror: 1  # 1 表示开启镜像，0 表示关闭，也可用 true 和 false 来表示
  # 剪裁一个 227*227 的图块，在训练阶段随机剪裁，在测试阶段从中间裁剪
  crop_size: 227
}
```

1、数据来自于数据库（如 LevelDB 和 LMDB）

层类型（layer type）:Data

必须设置的参数：

source: 包含数据库的目录名称，如 examples/mnist/mnist_train_lmdb

batch_size: 每次处理的数据个数，如 64

可选的参数：

rand_skip: 在开始的时候，路过某个数据的输入。通常对异步的 SGD 很有用。

backend: 选择是采用 LevelDB 还是 LMDB, 默认是 LevelDB.

示例：

```
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    scale: 0.00390625
  }
  data_param {
    source: "examples/mnist/mnist_train_lmdb"
    batch_size: 64
    backend: LMDB
  }
}
```

2、数据来自于内存

层类型：MemoryData

必须设置的参数：

batch_size: 每一次处理的数据个数，比如 2

channels: 通道数

height: 高度

width: 宽度

示例：

```

layer {
  top: "data"
  top: "label"
  name: "memory_data"
  type: "MemoryData"
  memory_data_param {
    batch_size: 2
    height: 100
    width: 100
    channels: 1
  }
  transform_param {
    scale: 0.0078125
    mean_file: "mean.proto"
    mirror: false
  }
}

```

3、数据来自于 HDF5

层类型: HDF5Data

必须设置的参数:

source: 读取的文件名称

batch_size: 每一次处理的数据个数

示例:

```

layer {
  name: "data"
  type: "HDF5Data"
  top: "data"
  top: "label"
  hdf5_data_param {
    source: "examples/hdf5_classification/data/train.txt"
    batch_size: 10
  }
}

```

4、数据来自于图片

层类型: ImageData

必须设置的参数:

source: 一个文本文件的名称, 每一行给定一个图片文件的名称和标签 (label)

batch_size: 每一次处理的数据个数, 即图片数

可选参数:

rand_skip: 在开始的时候, 路过某个数据的输入。通常对异步的 SGD 很有用。

shuffle: 随机打乱顺序, 默认值为 false

new_height,new_width: 如果设置, 则将图片进行 resize

示例:

```

layer {

```

```

name: "data"
type: "ImageData"
top: "data"
top: "label"
transform_param {
  mirror: false
  crop_size: 227
  mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto"
}
image_data_param {
  source: "examples/_temp/file_list.txt"
  batch_size: 50
  new_height: 256
  new_width: 256
}
}

```

5、数据来源于 Windows

层类型: WindowData

必须设置的参数:

source: 一个文本文件的名称

batch_size: 每一次处理的数据个数, 即图片数

示例:

```

layer {
  name: "data"
  type: "WindowData"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  transform_param {
    mirror: true
    crop_size: 227
    mean_file: "data/ilsrvrc12/imagenet_mean.binaryproto"
  }
  window_data_param {
    source: "examples/finetune_pascal_detection/window_file_2007_trainval.txt"
    batch_size: 128
    fg_threshold: 0.5
    bg_threshold: 0.5
    fg_fraction: 0.25
    context_pad: 16
    crop_mode: "warp"
  }
}

```

}

二. 视觉层 (Vision Layers)及参数

本文只讲解视觉层 (Vision Layers)的参数, 视觉层包括 Convolution, Pooling, Local Response Normalization (LRN), im2col 等层。

1、Convolution 层:

就是卷积层, 是卷积神经网络 (CNN) 的核心层。

层类型: Convolution

lr_mult: 学习率的系数, 最终的学习率是这个数乘以 solver.prototxt 配置文件中的 **base_lr**。如果有两个 **lr_mult**, 则第一个表示权值的学习率, 第二个表示偏置项的学习率。一般偏置项的学习率是权值学习率的两倍。

在后面的 **convolution_param** 中, 我们可以设定卷积层的特有参数。

必须设置的参数:

num_output: 卷积核 (filter)的个数

kernel_size: 卷积核的大小。如果卷积核的长和宽不等, 需要用 **kernel_h** 和 **kernel_w** 分别设定

其它参数:

stride: 卷积核的步长, 默认为 1。也可以用 **stride_h** 和 **stride_w** 来设置。

pad: 扩充边缘, 默认为 0, 不扩充。扩充的时候是左右、上下对称的, 比如卷积核的大小为 5*5, 那么 **pad** 设置为 2, 则四个边缘都扩充 2 个像素, 即宽度和高度都扩充了 4 个像素, 这样卷积运算之后的特征图就不会变小。也可以通过 **pad_h** 和 **pad_w** 来分别设定。

weight_filler: 权值初始化。默认为 "constant", 值全为 0, 很多时候我们用 "xavier" 算法来进行初始化, 也可以设置为 "gaussian"

bias_filler: 偏置项的初始化。一般设置为 "constant", 值全为 0。

bias_term: 是否开启偏置项, 默认为 true, 开启

group: 分组, 默认为 1 组。如果大于 1, 我们限制卷积的连接操作在一个子集内。如果我们根据图像的通道来分组, 那么第 i 个输出分组只能与第 i 个输入分组进行连接。

输入: $n \times c_0 \times w_0 \times h_0$

输出: $n \times c_1 \times w_1 \times h_1$

其中, c_1 就是参数中的 **num_output**, 生成的特征图个数

$w_1 = (w_0 + 2 * pad - kernel_size) / stride + 1;$

$h_1 = (h_0 + 2 * pad - kernel_size) / stride + 1;$

如果设置 **stride** 为 1, 前后两次卷积部分存在重叠。如果设置 $pad = (kernel_size - 1) / 2$, 则运算后, 宽度和高度不变。

示例:

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  param {
    lr_mult: 1
  }
}
```

```

param {
  lr_mult: 2
}
convolution_param {
  num_output: 20
  kernel_size: 5
  stride: 1
  weight_filler {
    type: "xavier"
  }
  bias_filler {
    type: "constant"
  }
}
}

```

2、Pooling 层

也叫池化层，为了减少运算量和数据维度而设置的一种层。

层类型：Pooling

必须设置的参数：

`kernel_size`: 池化的核大小。也可以用 `kernel_h` 和 `kernel_w` 分别设定。

其它参数：

`pool`: 池化方法，默认为 MAX。目前可用的方法有 MAX, AVE, 或 STOCHASTIC

`pad`: 和卷积层的 `pad` 的一样，进行边缘扩充。默认为 0

`stride`: 池化的步长，默认为 1。一般我们设置为 2，即不重叠(步长=窗口大小)。也可以用 `stride_h` 和 `stride_w` 来设置。

示例：

```

layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3
    stride: 2
  }
}

```

pooling 层的运算方法基本是和卷积层是一样的。

输入： $n \times c \times w_0 \times h_0$

输出： $n \times c \times w_1 \times h_1$

和卷积层的区别就是其中的 c 保持不变

$w_1 = (w_0 + 2 \times \text{pad} - \text{kernel_size}) / \text{stride} + 1$;

$h_1 = (h_0 + 2 \times \text{pad} - \text{kernel_size}) / \text{stride} + 1$;

如果设置 `stride` 为 2，前后两次卷积部分重叠。

3、Local Response Normalization (LRN)层

此层是对一个输入的局部区域进行归一化，达到“侧抑制”的效果。可去搜索 AlexNet 或 GoogLeNet，里面就用到了这个功能

层类型：LRN

参数：全部为可选，没有必须

local_size: 默认为 5。如果是跨通道 LRN，则表示求和的通道数；如果是在通道内 LRN，则表示求和的正方形区域长度。

alpha: 默认为 1，归一化公式中的参数。

beta: 默认为 5，归一化公式中的参数。

norm_region: 默认为 ACROSS_CHANNELS。有两个选择，ACROSS_CHANNELS 表示在相邻的通道间求和归一化。WITHIN_CHANNEL 表示在一个通道内部特定的区域内进行求和归一化。与前面的 local_size 参数对应。

归一化公式：对于每一个输入，去除以 $(1 + (\alpha/n) \sum_i x_i^2)^\beta$ ，得到归一化后的输出

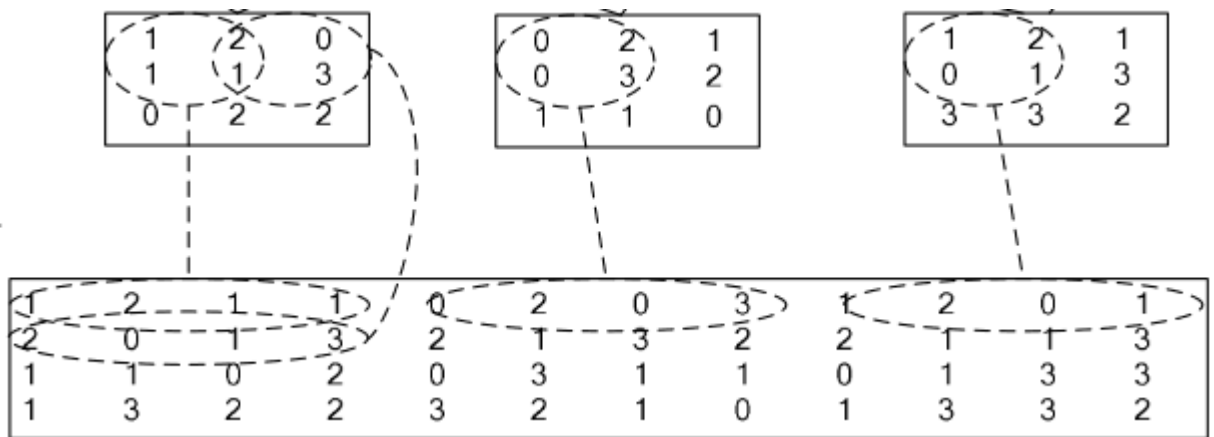
示例：

```
layers {  
  name: "norm1"  
  type: LRN  
  bottom: "pool1"  
  top: "norm1"  
  lrn_param {  
    local_size: 5  
    alpha: 0.0001  
    beta: 0.75  
  }  
}
```

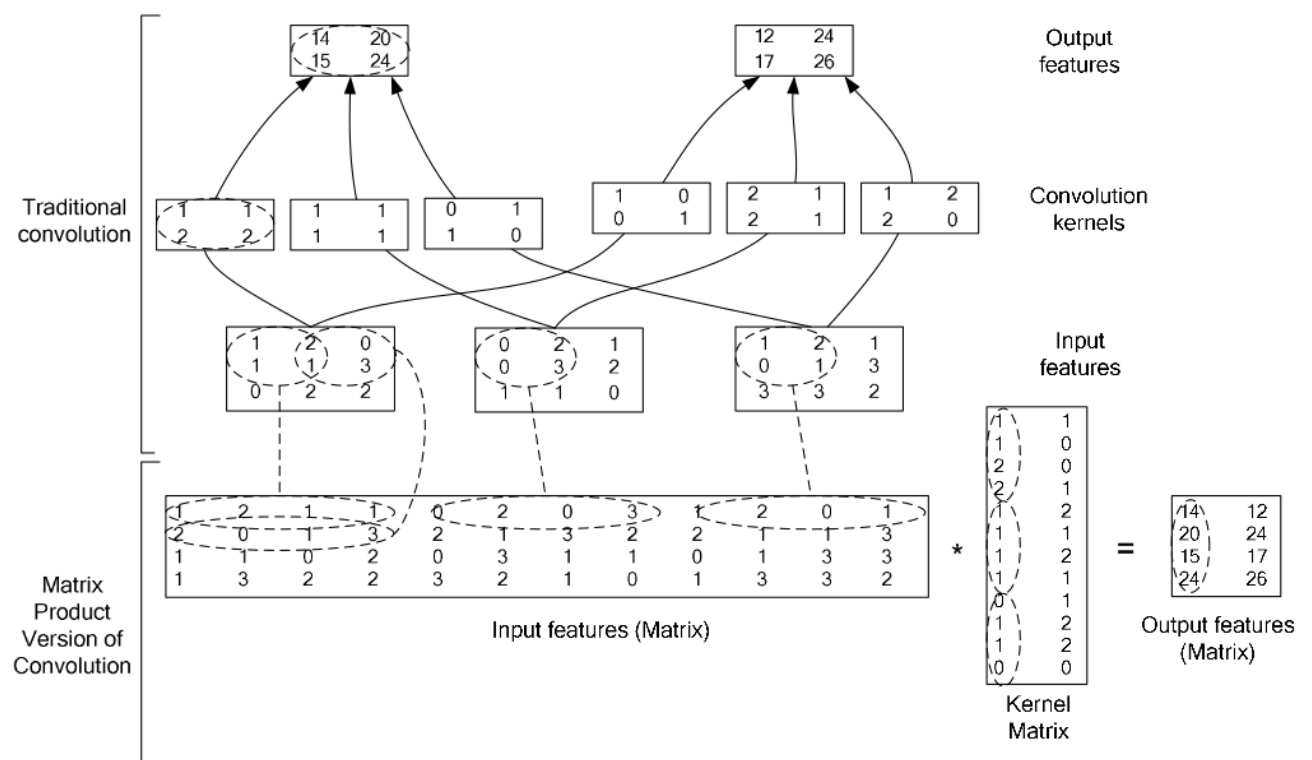
4、im2col 层

如果对 matlab 比较熟悉的话，就应该知道 im2col 是什么意思。它先将一个大矩阵，重叠地划分为多个子矩阵，对每个子矩阵序列化成为向量，最后得到另外一个矩阵。

看一看图就知道了：



在 caffe 中，卷积运算就是先对数据进行 im2col 操作，再进行内积运算（inner product）。这样做，比原始的卷积操作速度更快。
看看两种卷积操作的异同：



三. 激活层 (Activation Layers)及参数

在激活层中，对输入数据进行激活操作（实际上就是一种函数变换），是逐元素进行运算的。从 bottom 得到一个 blob 数据输入，运算后，从 top 输入一个 blob 数据。在运算过程中，没有改变数据的大小，即输入和输出的数据大小是相等的。

输入：n*c*h*w

输出：n*c*h*w

常用的激活函数有 sigmoid, tanh, relu 等，下面分别介绍。

1、Sigmoid

对每个输入数据，利用 sigmoid 函数执行操作。这种层设置比较简单，没有额外的参数。

$$S(x) = \frac{1}{1 + e^{-x}}$$

层类型: Sigmoid

示例:

```
layer {
  name: "encode1neuron"
  bottom: "encode1"
  top: "encode1neuron"
  type: "Sigmoid"
}
```

2、ReLU / Rectified-Linear and Leaky-ReLU

ReLU 是目前使用最多的激活函数，主要因为其收敛更快，并且能保持同样效果。

标准的 ReLU 函数为 $\max(x, 0)$ ，当 $x > 0$ 时，输出 x ；当 $x \leq 0$ 时，输出 0

$f(x) = \max(x, 0)$

层类型: ReLU

可选参数:

negative_slope: 默认为 0. 对标准的 ReLU 函数进行变化，如果设置了这个值，那么数据为负数时，就不再设置为 0，而是用原始数据乘以 **negative_slope**

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "pool1"
  top: "pool1"
}
```

RELU 层支持 in-place 计算，这意味着 bottom 的输出和输入相同以避免内存的消耗。

3、TanH / Hyperbolic Tangent

利用双曲正切函数对数据进行变换。

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

层类型: TanH

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "TanH"
}
```

4、Absolute Value

求每个输入数据的绝对值。

f(x)=Abs(x)

层类型: AbsVal

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
}
```

```
    type: "AbsVal"
}
```

5、Power

对每个输入数据进行幂运算

$f(x) = (\text{shift} + \text{scale} * x)^{\text{power}}$

层类型: Power

可选参数:

power: 默认为 1

scale: 默认为 1

shift: 默认为 0

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "Power"
  power_param {
    power: 2
    scale: 1
    shift: 0
  }
}
```

6、BNLL

binomial normal log likelihood 的简称

$f(x) = \log(1 + \exp(x))$

层类型: BNLL

```
layer {
  name: "layer"
  bottom: "in"
  top: "out"
  type: "BNLL"
}
```

四. 其它常用层及参数

本文讲解一些其它的常用层, 包括: softmax_loss 层, Inner Product 层, accuracy 层, reshape 层和 dropout 层及其它们的参数配置。

1、softmax-loss

softmax-loss 层和 softmax 层计算大致是相同的。softmax 是一个分类器, 计算的是类别的概率 (Likelihood), 是 Logistic Regression 的一种推广。Logistic Regression 只能用于二分类, 而 softmax 可以用于多分类。

softmax 与 softmax-loss 的区别:

softmax 计算公式:

$$p_j = \frac{e^{o_j}}{\sum_k e^{o_k}}$$

而 softmax-loss 计算公式:

$$L = - \sum_j y_j \log p_j,$$

关于两者的区别更加具体的介绍,可参考: [softmax vs. softmax-loss](#)

用户可能最终目的就是得到各个类别的概率似然值,这个时候就只需要一个 Softmax 层,而不一定要进行 softmax-Loss 操作;或者是用户有通过其他方式已经得到了某种概率似然值,然后要做最大似然估计,此时则只需要后面的 softmax-Loss 而不需要前面的 Softmax 操作。因此提供两个不同的 Layer 结构比只提供一个合在一起的 Softmax-Loss Layer 要灵活许多。

不管是 softmax layer 还是 softmax-loss layer,都是没有参数的,只是层类型不同而也

softmax-loss layer: 输出 loss 值

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip1"
  bottom: "label"
  top: "loss"
}
```

softmax layer: 输出似然值

```
layers {
  bottom: "cls3_fc"
  top: "prob"
  name: "prob"
  type: "Softmax"
}
```

2、Inner Product

全连接层,把输入当作成一个向量,输出也是一个简单向量(把输入数据 blobs 的 width 和 height 全变为 1)。

输入: $n \times c_0 \times h \times w$

输出: $n \times c_1 \times 1 \times 1$

全连接层实际上也是一种卷积层,只是它的卷积核大小和原数据大小一致。因此它的参数基本和卷积层的参数一样。

层类型: InnerProduct

lr_mult: 学习率的系数,最终的学习率是这个数乘以 solver.prototxt 配置文件中的 base_lr。

如果有两个 lr_mult,则第一个表示权值的学习率,第二个表示偏置项的学习率。一般偏置项的学习率是权值学习率的两倍。

必须设置的参数:

num_output: 过滤器 (filter)的个数

其它参数：

weight_filler: 权值初始化。默认为"constant",值全为 0，很多时候我们用"xavier"算法来进行初始化，也可以设置为"gaussian"

bias_filler: 偏置项的初始化。一般设置为"constant",值全为 0。

bias_term: 是否开启偏置项，默认为 true, 开启

```
layer {
  name: "ip1"
  type: "InnerProduct"
  bottom: "pool2"
  top: "ip1"
  param {
    lr_mult: 1
  }
  param {
    lr_mult: 2
  }
  inner_product_param {
    num_output: 500
    weight_filler {
      type: "xavier"
    }
    bias_filler {
      type: "constant"
    }
  }
}
```

3、accuracy

输出分类（预测）精确度，只有 test 阶段才有，因此需要加入 include 参数。

层类型：Accuracy

```
layer {
  name: "accuracy"
  type: "Accuracy"
  bottom: "ip2"
  bottom: "label"
  top: "accuracy"
  include {
    phase: TEST
  }
}
```

4、reshape

在不改变数据的情况下，改变输入的维度。

层类型：Reshape

先来看例子

```
layer {
```

```

name: "reshape"
type: "Reshape"
bottom: "input"
top: "output"
reshape_param {
  shape {
    dim: 0  # copy the dimension from below
    dim: 2
    dim: 3
    dim: -1 # infer it from the other dimensions
  }
}
}

```

有一个可选的参数组 `shape`, 用于指定 `blob` 数据的各维的值（`blob` 是一个四维的数据：`n*c*w*h`）。

`dim:0` 表示维度不变，即输入和输出是相同的维度。

`dim:2` 或 `dim:3` 将原来的维度变成 2 或 3

`dim:-1` 表示由系统自动计算维度。数据的总量不变，系统会根据 `blob` 数据的其它三维来自自动计算当前维的维度值。

假设原数据为：`64*3*28*28`，表示 64 张 3 通道的 `28*28` 的彩色图片

经过 `reshape` 变换：

```

reshape_param {
  shape {
    dim: 0
    dim: 0
    dim: 14
    dim: -1
  }
}

```

输出数据为：`64*3*14*56`

5、Dropout

Dropout 是一个防止过拟合的 `trick`。可以随机让网络某些隐含层节点的权重不工作。

先看例子：

```

layer {
  name: "drop7"
  type: "Dropout"
  bottom: "fc7-conv"
  top: "fc7-conv"
  dropout_param {
    dropout_ratio: 0.5
  }
}
layer {
  name: "drop7"
  type: "Dropout"

```

```

    bottom: "fc7-conv"
    top: "fc7-conv"
    dropout_param {
        dropout_ratio: 0.5
    }
}

```

只需要设置一个 dropout_ratio 就可以了

五. Blob, Layer and Net 以及对应配置文件的编写

深度网络(net)是一个组合模型，它由许多相互连接的层(layers)组合而成。Caffe 就是组建深度网络的这样一种工具，它按照一定的策略，一层一层的搭建出自己的模型。它将所有的信息数据定义为 blobs, 从而进行便利的操作和通讯。Blob 是 caffe 框架中一种标准的数组，一种统一的内存接口，它详细描述了信息是如何存储的，以及如何在层之间通讯的。

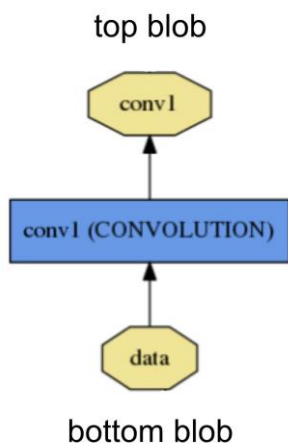
1、blob

Blobs 封装了运行时的数据信息，提供了 CPU 和 GPU 的同步。从数学上来说, Blob 就是一个 N 维数组。它是 caffe 中的数据操作基本单位，就像 matlab 中以矩阵为基本操作对象一样。只是矩阵是二维的，而 Blob 是 N 维的。N 可以是 2, 3, 4 等等。对于图片数据来说，Blob 可以表示为 (N*C*H*W) 这样一个 4D 数组。其中 N 表示图片的数量，C 表示图片的通道数，H 和 W 分别表示图片的高度和宽度。当然，除了图片数据，Blob 也可以用于非图片数据。比如传统的多层感知机，就是比较简单的全连接网络，用 2D 的 Blob，调用 innerProduct 层来计算就可以了。

在模型中设定的参数，也是用 Blob 来表示和运算。它的维度会根据参数的类型不同而不同。比如：在一个卷积层中，输入一张 3 通道图片，有 96 个卷积核，每个核大小为 11*11，因此这个 Blob 是 96*3*11*11。而在一个全连接层中，假设输入 1024 通道图片，输出 1000 个数据，则 Blob 为 1000*1024

2、layer

层是网络模型的组成要素和计算的基本单位。层的类型比较多，如 Data, Convolution, Pooling, ReLU, Softmax-loss, Accuracy 等，一个层的定义大至如下图：



从 bottom 进行数据的输入，计算后，通过 top 进行输出。图中的黄色多边形表示输入输出的数据，蓝色矩形表示层。

每一种类型的层都定义了三种关键的计算：setup, forward and backward

setup: 层的建立和初始化，以及在整个模型中的连接初始化。

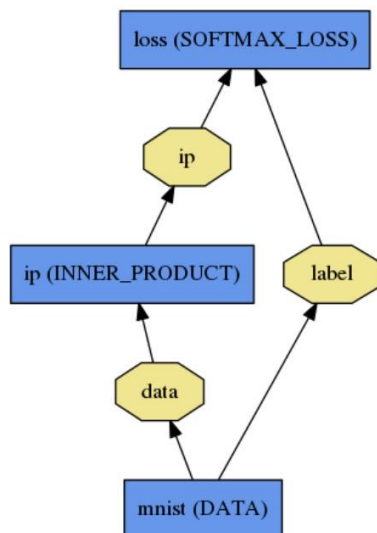
forward: 从 bottom 得到输入数据, 进行计算, 并将计算结果送到 top, 进行输出。

backward: 从层的输出端 top 得到数据的梯度, 计算当前层的梯度, 并将计算结果送到 bottom, 向前传递。

3、Net

就像搭积木一样, 一个 net 由多个 layer 组合而成。

现给出 一个简单的 2 层神经网络的模型定义(加上 loss 层就变成三层了), 先给出这个网络的拓扑。



第一层: name 为 mnist, type 为 Data, 没有输入 (bottom), 只有两个输出 (top), 一个为 data, 一个为 label

第二层: name 为 ip, type 为 InnerProduct, 输入数据 data, 输出数据 ip

第三层: name 为 loss, type 为 SoftmaxWithLoss, 有两个输入, 一个为 ip, 一个为 label, 有一个输出 loss, 没有画出来。

对应的配置文件 prototxt 就可以这样写:

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
```



```

        num_output: 2
    }
}
layer {
    name: "loss"
    type: "SoftmaxWithLoss"
    bottom: "ip"
    bottom: "label"
    top: "loss"
}

```

第一行将这个模型取名为 **LogReg**，然后是三个 **layer** 的定义，参数都比较简单，只列出必须的参数。

六. Solver 及其配置

solver 算是 **caffe** 的核心的核心，它协调着整个模型的运作。**caffe** 程序运行必带的一个参数就是 **solver** 配置文件。运行代码一般为

```
# caffe train --solver=*_solver.prototxt
```

在 Deep Learning 中，往往 **loss function** 是非凸的，没有解析解，我们需要通过优化方法来求解。**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，从而最小化 **loss**，实际上就是一种迭代的优化算法。

到目前的版本，**caffe** 提供了六种优化算法来求解最优参数，在 **solver** 配置文件中，通过设置 **type** 类型来选择。

Stochastic Gradient Descent (type: "SGD"),

AdaDelta (type: "AdaDelta"),

Adaptive Gradient (type: "AdaGrad"),

Adam (type: "Adam"),

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

RMSprop (type: "RMSProp")



具体的每种方法的介绍，请看本系列的下一篇文章，本文着重介绍 **solver** 配置文件的编写。

Solver 的流程:

1. 设计好需要优化的对象，以及用于学习的训练网络和用于评估的测试网络。（通过调用另外一个配置文件 **prototxt** 来进行）

2. 通过 **forward** 和 **backward** 迭代的进行优化来更新参数。

3. 定期的评价测试网络。（可设定多少次训练后，进行一次测试）

4. 在优化过程中显示模型和 **solver** 的状态

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 **solver** 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。

接下来，我们先来看一个实例：

```
net: "examples/mnist/lenet_train_test.prototxt"
```

```
test_iter: 100
```


test_interval: 500
base_lr: 0.01
momentum: 0.9
type: SGD
weight_decay: 0.0005
lr_policy: "inv"
gamma: 0.0001
power: 0.75
display: 100
max_iter: 20000
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet"
solver_mode: CPU

接下来，我们对每一行进行详细解译：

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。每一个 layer 的具体配置方式可参考本系列文文章中的（2）-（5）。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。例如：

train_net: "examples/hdf5_classification/logreg_auto_train.prototxt"

test_net: "examples/hdf5_classification/logreg_auto_test.prototxt"

接下来第二行：

test_iter: 100

这个要与 test layer 中的 batch_size 结合起来理解。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

测试间隔。也就是每训练 500 次，才进行一次测试。

base_lr: 0.01

lr_policy: "inv"

gamma: 0.0001

power: 0.75

这四行可以放在一起理解，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

$\text{base_lr} * \text{gamma}^{\text{floor}(\text{iter} / \text{stepsize})}$, 其中 iter 表示当前的迭代次数

- exp: 返回 $\text{base_lr} * \text{gamma}^{\text{iter}}$, 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 $\text{base_lr} * (1 + \text{gamma} * \text{iter})^{-\text{power}}$

- 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))))`

`multistep` 示例:

`base_lr: 0.01`

`momentum: 0.9`

`weight_decay: 0.0005`

`# The learning rate policy`

`lr_policy: "multistep"`

`gamma: 0.9`

`stepvalue: 5000`

`stepvalue: 7000`

`stepvalue: 8000`

`stepvalue: 9000`

`stepvalue: 9500`

接下来的参数:

`momentum : 0.9`

上一次梯度更新的权重，具体可参看下一篇文章。

`type: SGD`

优化算法选择。这一行可以省掉，因为默认值就是 `SGD`。总共有六种方法可选择，在本文的开头已介绍。

`weight_decay: 0.0005`

权重衰减项，防止过拟合的一个参数

`display: 100`

每训练 100 次，在屏幕上显示一次。如果设置为 0，则不显示。

`max_iter: 20000`

最大迭代次数。这个数设置太小，会导致没有收敛，精确度很低。设置太大，会导致震荡，浪费时间。

`snapshot: 5000`

`snapshot_prefix: "examples/mnist/lenet"`

快照。将训练出来的 `model` 和 `solver` 状态进行保存，`snapshot` 用于设置训练多少次后进行保存，默认为 0，不保存。`snapshot_prefix` 设置保存路径。

还可以设置 `snapshot_diff`，是否保存梯度值，默认为 `false`，不保存。

也可以设置 `snapshot_format`，保存的类型。有两种选择：`HDF5` 和 `BINARYPROTO`，默认为 `BINARYPROTO`

`solver_mode: CPU`

设置运行模式。默认为 `GPU`，如果你没有 `GPU`，则需要改成 `CPU`，否则会出错。

注意：以上的所有参数都是可选参数，都有默认值。根据 `solver` 方法 (`type`) 的不同，还有一些其它的参数，在此不一一列举。

七. Solver 优化方法

上文提到，到目前为止，`caffe` 总共提供了六种优化方法：

`Stochastic Gradient Descent (type: "SGD")`,

`AdaDelta (type: "AdaDelta")`,

Adaptive Gradient (type: "AdaGrad"),
Adam (type: "Adam"),
Nesterov's Accelerated Gradient (type: "Nesterov") and
RMSprop (type: "RMSProp")

Solver 就是用来使 loss 最小化的优化方法。对于一个数据集 D ，需要优化的目标函数是整个数据集中所有数据 loss 的平均值。

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

其中， $f_W(x(i))$ 计算的是数据 $x(i)$ 上的 loss，先将每个单独的样本 x 的 loss 求出来，然后求和，最后求均值。 $r(W)$ 是正则项 (weight_decay)，为了减弱过拟合现象。如果采用这种 Loss 函数，迭代一次需要计算整个数据集，在数据集非常大的这情况下，这种方法的效率很低，这个也是我们熟知的梯度下降采用的方法。

在实际中，通过将整个数据集分成几批 (batches)，每一批就是一个 mini-batch，其数量 (batch_size) 为 $N \ll |D|$ ，此时的 loss 函数为：

$$L(W) \approx \frac{1}{N} \sum_i^N f_W(X^{(i)}) + \lambda r(W)$$

有了 loss 函数后，就可以迭代的求解 loss 和梯度来优化这个问题。在神经网络中，用 forward pass 来求解 loss，用 backward pass 来求解梯度。

在 caffe 中，默认采用的 Stochastic Gradient Descent (SGD) 进行优化求解。后面几种方法也是基于梯度的优化方法 (like SGD)，因此本文只介绍一下 SGD。其它的方法，有兴趣的同学，可以去看文献原文。

1、Stochastic gradient descent (SGD)

随机梯度下降 (Stochastic gradient descent) 是在梯度下降法 (gradient descent) 的基础上发展起来的，梯度下降法也叫最速下降法，具体原理在网易公开课《机器学习》中，吴恩达教授已经讲解得非常详细。SGD 在通过负梯度 $-\nabla L(W)$ 和上一次的权重更新值 V_t 的线性组合来更新 W ，迭代公式如下：

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t)$$
$$W_{t+1} = W_t + V_{t+1}$$

其中， α 是负梯度的学习率(base_lr)， μ 是上一次梯度值的权重 (momentum)，用来加权之前梯度方向对现在梯度下降方向的影响。这两个参数需要通过 tuning 来得到最好的结果，一般是根据经验设定的。如果你不知道如何设定这些参数，可以参考相关的论文。在深度学习中使用 SGD，比较好的初始化参数的策略是把学习率设为 0.01 左右 (base_lr: 0.01)，在训练的过程中，如果 loss 开始出现稳定水平时，对学习率乘以一个常数因子 (gamma)，这样的过程重复多次。对于 momentum，一般取值在 0.5--0.99 之间。通常设为 0.9，momentum 可以让使用 SGD 的深度学习方法更加稳定以及快速。

关于更多的 momentum，请参看 Hinton 的

《A Practical Guide to Training Restricted Boltzmann Machines》。

实例：

```
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 1000
max_iter: 3500
momentum: 0.9
```

lr_policy 设置为 step,则学习率的变化规则为 $\text{base_lr} * \text{gamma}^{\lfloor \text{iter} / \text{stepsize} \rfloor}$

即前 1000 次迭代，学习率为 0.01；第 1001-2000 次迭代，学习率为 0.001；第 2001-3000 次迭代，学习率为 0.00001，第 3001-3500 次迭代，学习率为 10^{-5}

上面的设置只能作为一种指导，它们不能保证在任何情况下都能得到最佳的结果，有时候这种方法甚至不 work。如果学习的时候出现 diverge（比如，你一开始就发现非常大或者 NaN 或者 inf 的 loss 值或者输出），此时你需要降低 base_lr 的值（比如，0.001），然后重新训练，这样的过程重复几次直到你找到可以 work 的 base_lr。

2、AdaDelta

AdaDelta 是一种“鲁棒的学习率方法”，是基于梯度的优化方法（like SGD）。

具体的介绍文献：

M. Zeiler ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. arXiv preprint, 2012.

示例：

```
net: "examples/mnist/lenet_train_test.prototxt"
test_iter: 100
test_interval: 500
base_lr: 1.0
lr_policy: "fixed"
momentum: 0.95
weight_decay: 0.0005
display: 100
max_iter: 10000
```

```
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet_adadelta"
solver_mode: GPU
type: "AdaDelta"
delta: 1e-6
```

从最后两行可看出，设置 solver type 为 Adadelta 时，需要设置 delta 的值。

3、AdaGrad

自适应梯度（adaptive gradient）是基于梯度的优化方法（like SGD）

具体的介绍文献：

Duchi, E. Hazan, and Y. Singer. **Adaptive Subgradient Methods for Online Learning and Stochastic Optimization**. The Journal of Machine Learning Research, 2011.

示例：

```
net: "examples/mnist/mnist_autoencoder.prototxt"
test_state: { stage: 'test-on-train' }
test_iter: 500
test_state: { stage: 'test-on-test' }
test_iter: 100
test_interval: 500
test_compute_loss: true
base_lr: 0.01
lr_policy: "fixed"
display: 100
max_iter: 65000
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "examples/mnist/mnist_autoencoder_adagrad_train"
# solver mode: CPU or GPU
solver_mode: GPU
type: "AdaGrad"
```

4、Adam

是一种基于梯度的优化方法（like SGD）。

具体的介绍文献：

D. Kingma, J. Ba. **Adam: A Method for Stochastic Optimization**. International Conference for Learning Representations, 2015.

5、NAG

Nesterov 的加速梯度法（Nesterov's accelerated gradient）作为凸优化中最理想的方法，其收敛速度非常快。

具体的介绍文献：

I. Sutskever, J. Martens, G. Dahl, and G. Hinton. **On the Importance of Initialization and Momentum in Deep Learning**. Proceedings of the 30th International Conference on Machine Learning, 2013.

示例:

```
net: "examples/mnist/mnist_autoencoder.prototxt"
test_state: { stage: 'test-on-train' }
test_iter: 500
test_state: { stage: 'test-on-test' }
test_iter: 100
test_interval: 500
test_compute_loss: true
base_lr: 0.01
lr_policy: "step"
gamma: 0.1
stepsize: 10000
display: 100
max_iter: 65000
weight_decay: 0.0005
snapshot: 10000
snapshot_prefix: "examples/mnist/mnist_autoencoder_nesterov_train"
momentum: 0.95
# solver mode: CPU or GPU
solver_mode: GPU
type: "Nesterov"
```

6、RMSprop

RMSprop 是 Tieleman 在一次 Coursera 课程演讲中提出来的, 也是一种基于梯度的优化方法 (like SGD)

具体的介绍文献:

T. Tieleman, and G. Hinton. **RMSProp: Divide the gradient by a running average of its recent magnitude**. COURSERA: Neural Networks for Machine Learning. Technical report, 2012.

示例:

```
net: "examples/mnist/lenet_train_test.prototxt"
test_iter: 100
test_interval: 500
base_lr: 1.0
lr_policy: "fixed"
momentum: 0.95
weight_decay: 0.0005
display: 100
max_iter: 10000
```

```
snapshot: 5000
snapshot_prefix: "examples/mnist/lenet_adadelta"
solver_mode: GPU
type: "RMSProp"
rms_decay: 0.98
```

最后两行，需要设置 rms_decay 值。

八. 命令行解析

caffe 的运行提供三种接口：c++接口（命令行）、Python 接口和 matlab 接口。本文先对命令行进行解析，后续会依次介绍其它两个接口。

caffe 的 c++主程序 (caffe.cpp)放在根目录下的 tools 文件夹内，当然还有一些其它的功能文件，如：convert_imageset.cpp, train_net.cpp, test_net.cpp 等也放在这个文件夹内。经过编译后，这些文件都被编译成了可执行文件，放在了 ./build/tools/ 文件夹内。因此我们要执行 caffe 程序，都需要加 ./build/tools/ 前缀。

如：

```
# sudo sh ./build/tools/caffe train
--solver=examples/mnist/train_lenet.sh
```

caffe 程序的命令行执行格式如下：

```
caffe <command> <args>
```

其中的<command>有这样四种：

- train
- test
- device_query
- time

对应的功能为：

train----训练或 finetune 模型 (model),

test----测试模型

device_query---显示 gpu 信息

time-----显示程序执行时间

其中的<args>参数有：

- -solver
- -gpu
- -snapshot
- -weights
- -iteration

- -model
- -sighup_effect
- -sigint_effect

注意前面有个-符号。对应的功能为：

-solver: 必选参数。一个 protocol buffer 类型的文件，即模型的配置文件。如：

```
# ./build/tools/caffe train -solver
examples/mnist/lenet_solver.prototxt
```

-gpu: 可选参数。该参数用来指定用哪一块 gpu 运行，根据 gpu 的 id 进行选择，如果设置为'-gpu all'则使用所有的 gpu 运行。如使用第二块 gpu 运行：

```
# ./build/tools/caffe train -solver
examples/mnist/lenet_solver.prototxt -gpu 2
```

-snapshot:可选参数。该参数用来从快照 (snapshot)中恢复训练。可以在 solver 配置文件设置快照，保存 solverstate。如：

```
# ./build/tools/caffe train -solver
examples/mnist/lenet_solver.prototxt -snapshot
examples/mnist/lenet_iter_5000.solverstate
```

-weights:可选参数。用预先训练好的权重来 fine-tuning 模型，需要一个 caffemodel，不能和-snapshot 同时使用。如：

```
# ./build/tools/caffe train -solver
examples/finetuning_on_flickr_style/solver.prototxt -weights
models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel
```

-iterations: 可选参数，迭代次数，默认为 50。如果在配置文件文件中没有设定迭代次数，则默认迭代 50 次。

-model:可选参数，定义在 protocol buffer 文件中的模型。也可以在 solver 配置文件中指定。

-sighup_effect: 可选参数。用来设定当程序发生挂起事件时，执行的操作，可以设置为 snapshot, stop 或 none, 默认为 snapshot

-sigint_effect: 可选参数。用来设定当程序发生键盘中止事件时 (ctrl+c), 执行的操作，可以设置为 snapshot, stop 或 none, 默认为 stop

刚才举例了一些 train 参数的例子，现在来看看其它三个<command>:

test 参数用在测试阶段，用于最终结果的输出，要模型配置文件中我们可以设定需要输入 accuracy 还是 loss. 假设我们要在验证集中验证已经训练好的模型，就可以这样写

```
# ./build/tools/caffe test -model
examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 100
```



这个例子比较长，不仅用到了 **test** 参数，还用到了 **-model**, **-weights**, **-gpu** 和 **-iteration** 四个参数。意思是利用训练好了的权重 (**-weight**)，输入到测试模型中(**-model**)，用编号为 0 的 **gpu(-gpu)**测试 100 次(**-iteration**)。

time 参数用来在屏幕上显示程序运行时间。如：

```
# ./build/tools/caffe time -model
examples/mnist/lenet_train_test.prototxt -iterations 10
```

这个例子用来在屏幕上显示 **lenet** 模型迭代 10 次所使用的时间。包括每次迭代的 **forward** 和 **backward** 所用的时间，也包括每层 **forward** 和 **backward** 所用的平均时间。

```
# ./build/tools/caffe time -model
examples/mnist/lenet_train_test.prototxt -gpu 0
```

这个例子用来在屏幕上显示 **lenet** 模型用 **gpu** 迭代 50 次所使用的时间。

```
# ./build/tools/caffe time -model
examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 10
```

利用给定的权重，利用第一块 **gpu**，迭代 10 次 **lenet** 模型所用的时间。

device_query 参数用来诊断 **gpu** 信息。

```
# ./build/tools/caffe device_query -gpu 0
```

最后，我们来看两个关于 **gpu** 的例子

```
# ./build/tools/caffe train -solver
examples/mnist/lenet_solver.prototxt -gpu 0,1
# ./build/tools/caffe train -solver
examples/mnist/lenet_solver.prototxt -gpu all
```

这两个例子表示： 用两块或多块 **GPU** 来平行运算，这样速度会快很多。但是如果你只有一块或没有 **gpu**，就不要加 **-gpu** 参数了，加了反而慢。

最后，在 **linux** 下，本身就有一个 **time** 命令，因此可以结合进来使用，因此我们运行 **mnist** 例子的最终命令是(一块 **gpu**)：

```
$ sudo time ./build/toos/caffe train -solver
examples/mnist/lenet_solver.prototxt
```