Caffe 官方教程中译本

社区预览版 V1.0

CaffeCN 深度学习社区志愿者集体翻译

http://caffecn.cn | QQ group: 431141753 | @CaffeCN 深度学习社区

目 录

致谢：

CaffeCN 社区谨此向 Caffe 官方社区致敬！

向所有参加了此次文档翻译工作的志愿者 [1] 致以诚挚的感谢！

CaffeCN 社区

2016.02.01

[1] 翻译和校对人员包括：

巴君、戴嘉伦、龚国平、李文杰、刘畅、刘昕、刘艳飞、马杰超、任伟、师亚亭、孙琳钧、它它、王斌、王洪振、王蒙蒙、吴尚轩、辛淼、杨轶斐、占鸿渐、张欣、赵行、郑昌艳。

前言

Caffe 是一个深度学习框架。本教程讲述了 Caffe 的设计哲学、架构和使用方法。这是一份关于 Caffe 的实践指引和介绍，但文中并不包含关于深度学习的前沿、进展和历史等方面的内容。尽管我们会在必要的地方给出一些辅助解释，但如果读者具有关于机器学习和神经网络方面的背景知识，将十分有助于理解本文的内容。

设计哲学

简言之，Caffe 的酝酿考虑了以下几个方面的内容：

- 表示：模型和优化以纯文本的模式定义，而不是以代码模式；
- 速度：对于学术领域和工业领域，运算速度对于最先进的模型和海量数据是至关重要的；
- 模块化：新的任务和配置要求框架具有灵活性和扩展性；
- 开放性：科研和应用过程需要公共的代码、可参考的模型和可再现性；
- 社区：通过共同讨论和以 BSD-2 协议共同开发这个项目，学术研究、起步阶段的原型和工业应用可以共享各自的力量。

这些准则指引了整个项目。

教程内容

- Nets, Layers, and Blobs：Caffe 模型解析；
- Forward and Backward：层状模型的基本计算；
- Loss：由 loss 定义待学习的任务；
- Solver：solver 协调模型的优化；
- Layer Catalogue：“层”是模型和计算的基本单元，Caffe 提供的结构中包含了构建先进模型所需的各种层；
- Interface：Caffe 的命令行，Python，和 MATLAB 版接口；
- Data：如何为模型添加 caffe 式的输入数据。

第一章 Blobs, Layers, and Nets：Caffe 模型解析

深度神经网络是一种模块化的模型，它由一系列作用在数据块之上的内部连接层组合而成。Caffe 基于自己的模型架构，通过逐层定义（layer-by-layer）的方式定义一个网络（Nets）。网络从数据输入层到损失层自下而上地定义整个模型。Caffe 使用 *blobs* 结构来存储、交换和处理网络中正向和反向迭代时的数据和导数信息：blob 是 Caffe 的标准数组结构，它提供了一个统一的内存接口。Layer 是 Caffe 模型和计算的基本单元，Net 是一系列 layers 和其连接的集合。Blob 详细描述了信息是如何在 layer 和 net 中存储和交换的。

Solving（求解方法）单独配置，以解耦模型的建立与优化的过程。

下面将详细介绍这些组成部分。

1.1 Blob 的存储与交换

Blob 是 Caffe 中处理和传递实际数据的数据封装包，并且在 CPU 与 GPU 之间具有同步处理能力。从数学意义上说，blob 是按 C 风格连续存储的 N 维数组。

Caffe 基于 blobs 存储和交换数据。为了便于优化，blobs 提供统一的内存接口来存储某种类型的数据，例如批量图像数据、模型参数以及用来进行优化的导数。

Blobs 可根据 CPU 主机到 GPU 设备的同步需要，屏蔽 CPU/GPU 混和运算在计算上的开销。主机和设备上的内存按需求分配（lazily），以提高内存的使用效率。

对于批量图像数据来说，blob 常规的维数为图像数量 N *通道数 K *图像高度 H *图像宽度 W。Blob 按行为主（row-major）进行存储，所以一个 4 维 blob 中，坐标为(n, k, h, w)的值的物理位置为（(n * K + k) * H + h) * W + w，这也使得最后面/最右边的维度更新最快。

- Number/N 是每个批次处理的数据量。批量处理信息有利于提高设备处理和交换的数据的吞吐率。在 ImageNet 上每个训练批量为 256 张图像，则 N=256。
- Channel/K 是特征维度，例如对 RGB 图像来说，K=3.

虽然 Caffe 的图像应用例子中很多 blobs 都是 4 维坐标，但是对于非图像应用任务，blobs 也完全可以照常使用。例如，如果你仅仅需要类似于传统多层感知机那样的全连接网络，使用 2 维的 blobs（形式为(N, D)），之后再调用 InnerProductLayer（全连接层，随后我们将讲解）即可。

参数 Blob 的维度是根据层的类型和配置而变化的。一个卷积层中若有 96 个空间维度为

11 x 11、输入为 3 通道的滤波器，那么其 blob 维度是 96 x 3 x 11 x 11。对于一个输入是 1024 维（输入通道数），输出是 1000 维（输出通道数）的内积层/全连接层，参数 blob 维度是 1000 x 1024。

对于一些特定数据，自己设计输入工具或者数据层是很有必要的。但是无论怎样，一旦你的数据准备完毕，各种层模块将会帮你完成剩下的工作。

1.1.1 实现细节

对于 blob 中的数据，我们关心的是 values（值）和 gradients（梯度），所以一个 blob 单元存储了两块数据——*data* 和 *diff*。前者是我们在网络中传送的普通数据，后者是通过网络计算得到的梯度。

而且，由于数据既可存储在 CPU 上，也可存储在 GPU 上，因而有两种数据访问方式：静态方式，不改变数值；动态方式，改变数值。

```
const Dtype* cpu_data() const;

Dtype* mutable_cpu_data();
```

（gpu 和 diff 的操作与之类似）

之所以这么设计是因为 blob 使用了一个 SyncedMem 类来同步 CPU 和 GPU 上的数值，以隐藏同步的细节和最小化传送数据。一个经验准则是，如果不想改变数值，就一直使用常量调用，而且绝不要在自定义类中存储指针。每次操作 blob 时，调用相应的函数来获取它的指针，因为 SyncedMem 需要用这种方式来确定何时需要复制数据。

实际上，使用 GPU 时，Caffe 中 CPU 代码先从磁盘中加载数据到 blob，同时请求分配一个 GPU 设备核（device kernel）以使用 GPU 进行计算，再将计算好的 blob 数据送入下一层，这样既实现了高效运算，又忽略了底层细节。只要所有 layers 均有 GPU 实现，这种情况下所有的中间数据和梯度都会保留在 GPU 上。

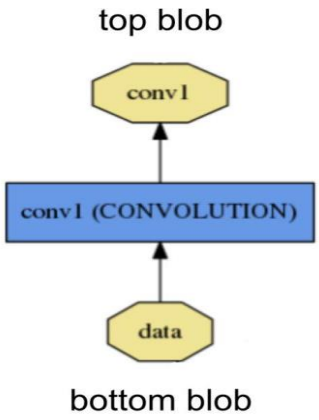
这里有一个示例，用以确定 blob 何时会复制数据：

```
// 假定数据在 CPU 上进行初始化，我们有一个 blob
const Dtype* foo;

Dtype* bar;

foo = blob.gpu_data(); // 数据从 CPU 复制到 GPU
```

```
foo = blob.cpu_data(); // 没有数据复制，两者都有最新的内容
bar = blob.mutable_gpu_data(); // 没有数据复制
// ... 一些操作 ...
bar = blob.mutable_gpu_data(); // 仍在 GPU，没有数据复制
foo = blob.cpu_data(); // 由于 GPU 修改了数值，数据从 GPU 复制到 CPU
foo = blob.gpu_data(); //没有数据复制，两者都有最新的内容
bar = blob.mutable_cpu_data(); // 依旧没有数据复制
bar = blob.mutable_gpu_data(); //数据从 CPU 复制到 GPU
bar = blob.mutable_cpu_data(); //数据从 GPU 复制到 CPU
```

1.2 Layer 的计算和连接

Layer 是 Caffe 模型的本质内容和执行计算的基本单元。Layer 可以进行很多运算，如：convolve（卷积）、pool（池化）、inner product（内积），rectified-linear 和 sigmoid 等非线性运算，元素级的数据变换，normalize（归一化）、load data（数据加载）、softmax 和 hinge 等 losses（损失计算）。可在 Caffe 的 layer catalogue（层目录）中查看所有操作，其囊括了绝大部分目前最前沿的深度学习任务所需要的层类型。



一个 layer 通过 *bottom*（底部）连接层接收数据，通过 *top*（顶部）连接层输出数据。

每一个 layer 都定义了 3 种重要的运算：*setup*（初始化设置），*forward*（前向传播），*backward*（反向传播）。

- Setup: 在模型初始化时重置 layers 及其相互之间的连接；
- Forward: 从 bottom 层中接收数据，进行计算后将输出送入到 top 层中;
- Backward: 给定相对于 top 层输出的梯度，计算其相对于输入的梯度，并传递到 bottom

层。一个有参数的 layer 需要计算相对于各个参数的梯度值并存储在内部。

特别地，Forward 和 Backward 函数分别有 CPU 和 GPU 两种实现方式。如果没有实现 GPU 版本，那么 layer 将转向作为备用选项的 CPU 方式。尽管这样会增加额外的数据传送成本（输入数据由 GPU 上复制到 CPU，之后输出数据从 CPU 又复制回到 GPU），但是对于做一些快速实验这样操作还是很方便的。

总的来说，Layer 承担了网络的两个核心操作：*forward pass*（前向传播）——接收输入并计算输出；*backward pass*（反向传播）——接收关于输出的梯度，计算相对于参数和输入的梯度并反向传播给在它前面的层。由此组成了每个 layer 的前向和反向通道。

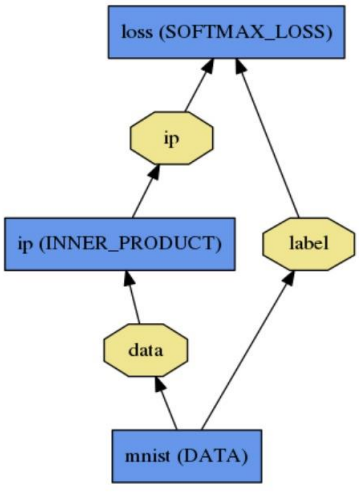
由于 Caffe 网络的组合性和其代码的模块化，自定义 layer 是很容易的。只要定义好 layer 的 setup（初始化设置）、forward（前向通道）和 backward（反向通道），就可将 layer 纳入到网络中。

1.3 Net 的定义和操作

通过合成和自动微分，网络同时定义了一个函数和其对应的梯度。通过合成各层的输出来计算这个函数，来执行给定的任务，并通过合成各层的后向传播过程来计算来自损失函数的梯度，从而学习任务。Caffe 模型是端到端的机器学习引擎。

准确的说，Net 是由一系列层组成的有向无环（DAG）计算图，Caffe 保留了计算图中所有的中间值以确保前向和反向迭代的准确性。一个典型的 Net 开始于 data layer——从磁盘中加载数据，终止于 loss layer——计算如分类和重构这些任务的目标函数。

Net 由一系列层和它们之间的相互连接构成，用的是一种文本建模语言。一个简单的逻辑回归分类器的定义如下：



```
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"
}
```

Net::Init()进行模型的初始化。初始化主要实现两个操作：创建 blobs 和 layers 以搭建整个网络 DAG 图（向一些 C++极客声明：我们将一直持有 blobs 和 layers 的所有权），以及调用 layers 的 **SetUp()**函数。初始化时也会做另一些记录，例如确认整个网络结构的正确与否等。另外，初始化期间，Net 会打印其初始化日志到 INFO 信息中。

```
I0902 22:52:17.931977 2079114000 net.cpp:39] Initializing net from parameters:
name: "LogReg"
[...model prototxt printout...]
# construct the network layer-by-layer
I0902 22:52:17.932152 2079114000 net.cpp:67] Creating Layer mnist
I0902 22:52:17.932165 2079114000 net.cpp:356] mnist -> data
I0902 22:52:17.932188 2079114000 net.cpp:356] mnist -> label
I0902 22:52:17.932200 2079114000 net.cpp:96] Setting up mnist
I0902 22:52:17.935807 2079114000 data_layer.cpp:135] Opening leveldb
input_leveldb
I0902 22:52:17.937155 2079114000 data_layer.cpp:195] output data size:
64,1,28,28
I0902 22:52:17.938570 2079114000 net.cpp:103] Top shape: 64 1 28 28 (50176)
I0902 22:52:17.938593 2079114000 net.cpp:103] Top shape: 64 (64)
I0902 22:52:17.938611 2079114000 net.cpp:67] Creating Layer ip
I0902 22:52:17.938617 2079114000 net.cpp:394] ip <- data
I0902 22:52:17.939177 2079114000 net.cpp:356] ip -> ip
I0902 22:52:17.939196 2079114000 net.cpp:96] Setting up ip
I0902 22:52:17.940289 2079114000 net.cpp:103] Top shape: 64 2 (128)
I0902 22:52:17.941270 2079114000 net.cpp:67] Creating Layer loss
I0902 22:52:17.941305 2079114000 net.cpp:394] loss <- ip
I0902 22:52:17.941314 2079114000 net.cpp:394] loss <- label
I0902 22:52:17.941323 2079114000 net.cpp:356] loss -> loss
# set up the loss and configure the backward pass
I0902 22:52:17.941328 2079114000 net.cpp:96] Setting up loss
```

```
I0902 22:52:17.941328 2079114000 net.cpp:103] Top shape: (1)
I0902 22:52:17.941329 2079114000 net.cpp:109]     with loss weight 1
I0902 22:52:17.941779 2079114000 net.cpp:170] loss needs backward computation.
I0902 22:52:17.941787 2079114000 net.cpp:170] ip needs backward computation.
I0902 22:52:17.941794 2079114000 net.cpp:172] mnist does not need backward
computation.
# determine outputs
I0902 22:52:17.941800 2079114000 net.cpp:208] This network produces output loss
# finish initialization and report memory usage
I0902 22:52:17.941810 2079114000 net.cpp:467] Collecting Learning Rate and Weight
Decay.
I0902 22:52:17.941818 2079114000 net.cpp:219] Network initialization done.
I0902 22:52:17.941824 2079114000 net.cpp:220] Memory required for data: 201476
```

Caffe 中网络的构建与设备无关，可回忆下我们之前的解释，blobs 和 layers 在模型定义时是隐藏了实现细节的。网络构建完之后，通过设置 **Caffe::mode()**函数中的 **Caffe::set_mode()**，即可实现在 CPU 或 GPU 上的运行。采用 CPU 或 GPU 计算得到的结果相同（通过多次实验已证明），CPU 与 GPU 无缝切换并且独立于模型定义。对于研究和调用来说，将模型定义和实现分离开来是最好不过了。

1.3.1 模型格式

模型是利用文本 protocol buffer（prototxt）语言定义的，学习好的模型会被序列化地存储在二进制 protocol buffer (binaryproto) .caffemodel 文件中。

模型格式用 protobuf 语言定义在 caffe.proto 文件中。大部分源文件中都带有解释，所以鼓励大家去查看。

Caffe 使用 Google Protocol Buffer 有以下优势：按序排列时二进制字符串尺寸最小，高效序列化，易读的文本格式与二进制版本兼容，可用多种语言实现高效的接口，尤其是 C++ 和 Python。这些优势造就了 Caffe 模型的灵活性与扩展性。

本章翻译、校对、审校人员：

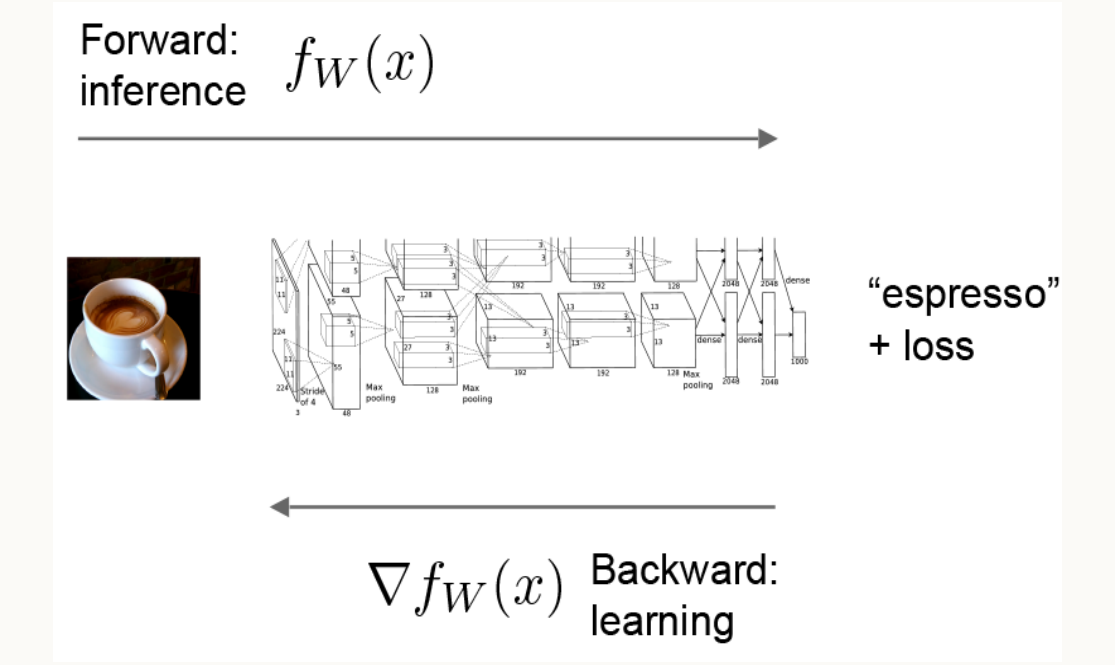
翻译：郑昌艳

校对：王蒙蒙、马杰超

审校：辛淼

第二章 Forward and Backward（前传/反传）

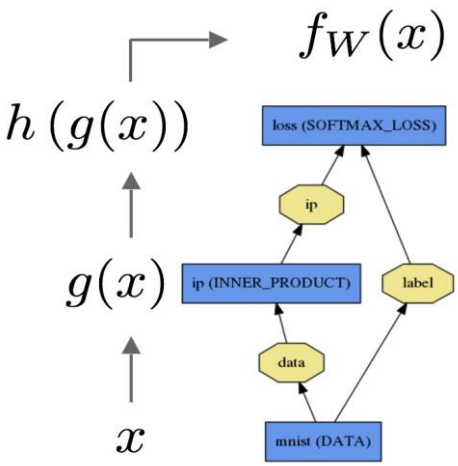
前传和反传是一个网络最重要的计算过程。



下面以最简单的逻辑回归分类器为例。

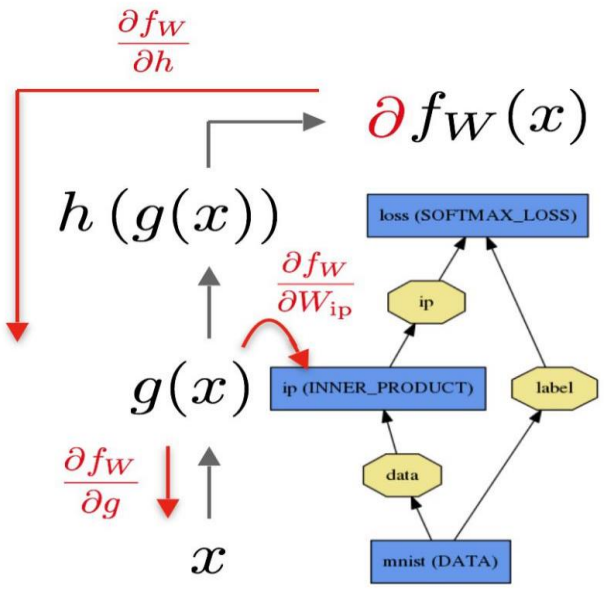
2.1 前传

前传（forward）过程为给定的待推断的输入计算输出。在前传过程中，Caffe 组合每一层的计算以得到整个模型的计算“函数”。本过程自底向上进行。数据 x 通过一个内积层得到 $g(x)$，然后通过 softmax 层得到 $h(g(x))$，通过 softmax loss 得到 $fw(x)$。



2.2 反传

反传（backward）过程根据损失来计算梯度从而进行学习。在反传过程中，Caffe 通过自动求导并反向组合每一层的梯度来计算整个网络的梯度。这就是反传过程的本质。本过程自顶向下进行。



反传过程以损失开始，然后根据输出计算梯度$\frac{\partial f_w}{\partial h}$。根据链式准则，逐层计算出模型其余部分的梯度。有参数的层，例如 INNER_PRODUCT 层，会在反传过程中根据参数计算梯度$\frac{\partial f_w}{\partial W_{ip}}$。

2.3 Caffe 中前传和反传的实现

只要定义好了模型，这些计算就可以立即进行：Caffe 已经为你准备好了前传和反传的实现方法。

- Net::Forward()和 Net::Backward()方法实现网络的前传和后传，而 Layer::Forward()和 Layer::Backward()计算每一层的前传后传。
- 每一层都有 forward_{cpu, gpu}()和 backward_{cpu, gpu}方法来适应不同的计算模式。由于条件限制或者为了使用便利，一个层可能仅实现了 CPU 或者 GPU 模式。

Solver 优化一个模型，首先通过调用前传来获得输出和损失，然后调用反传产生模型的梯度，将梯度与权值更新后相结合来最小化损失。Solver、网络和层之间的分工使得 Caffe 可以模块化并且开源。

若想了解更多关于 Caffe 前传和后传的层的类型，可参考层类别部分。

本章翻译、校对、审校人员：

翻译：师亚亭

校对：潘智斌、张欣

审校：辛淼

第三章 Loss

与大多数的机器学习模型一样，在 Caffe 中，学习是由一个**损失**函数驱动的（通常也被称为**误差**、**代价**或者**目标**函数）。一个损失函数通过将参数集（即当前的网络权值）映射到一个可以标识这些参数“不良程度”的标量值来学习目标。因此，学习的目的是找到一个网络权重的集合，使得损失函数最小。

在 Caffe 中，损失是通过网络的前向计算得到的。每一层由一系列的输入 blobs (*bottom*)，然后产生一系列的输出 blobs (*top*)。这些层的某些输出可以用来作为损失函数。典型的一对多分类任务的损失函数是 softMaxWithLoss 函数，使用以下的网络定义，例如:

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

在 softMaxWithLoss 函数中，*top* blob 是一个标量数值，该数值是整个 batch 的损失平均值（由预测值 pred 和真实值 label 计算得到）。

3.1 Loss weights

对于含有多个损失层的网络（例如，一个网络使用一个 softMaxWithLoss 输入分类并使用 EuclideanLoss 层进行重构），损失权值可以被用来指定它们之间的相对重要性。

按照惯例，有着 Loss 后缀的 Caffe 层对损失函数有贡献，其他层被假定仅仅用于中间计算。然而，通过在层定义中添加一个 loss_weight:<float>字段到由该层的 top blob，任何层都可以作为一个 loss。对于带后缀 Loss 的层来说，其对于该层的第一个 top blob 含有一个隐式的 loss_weight:1；其他层对应于所有 top blob 有一个隐式的 loss_weight:0。因此，上面的 softMaxWithLoss 层等价于：

```
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  loss_weight: 1
}
```

然而，任何可以反向传播的层，可允许给予一个非 0 的 loss_weight，例如，如果需要，对网络的某些中间层所产生的激活进行正则化。对于具有相关非 0 损失的非单输出，损失函数可以通过对所有 blob 求和来进行简单地计算。

那么，在 Caffe 中最终的损失函数可以通过对整个网络中所有的权值损失进行求和计算获得，正如以下的伪代码：

```
loss := 0
for layer in layers:
  for top, loss_weight in layer.tops, layer.loss_weights:
    loss += loss_weight * sum(top)
```

本章翻译、校对、审校人员：

翻译：龚国平

校对：刘艳飞

审校：辛淼

第四章 Solver

4.1 Solver 简介

Solver 通过协调 Net 的前向推断计算和反向梯度计算（forward inference and backward gradients），来对参数进行更新，从而达到减小 loss 的目的。Caffe 模型的学习被分为两个部分：由 Solver 进行优化、更新参数，由 Net 计算出 loss 和 gradient。

Caffe 支持的 solvers 包括:

- Stochastic Gradient Descent (type: "SGD")，随机梯度下降
- AdaDelta (type: "AdaDelta")
- Adaptive Gradient (type: "AdaGrad")，自适应梯度
- Adam (type: "Adam")
- Nesterov’s Accelerated Gradient (type: "Nesterov") and
- RMSprop (type: "RMSProp")

Solver：

1. 用于优化过程的记录、创建训练网络（用于学习）和测试网络（用于评估）；
2. 通过 forward 和 backward 过程来迭代地优化和更新参数；
3. 周期性地用测试网络评估模型性能；
4. 在优化过程中记录模型和 solver 状态的快照（snapshot）；

每一次迭代过称中：

1. 调用 Net 的前向过程计算出输出和 loss；

2. 调用 Net 的后向过程计算出梯度（loss 对每层的权重 w 和偏置 b 求导）；

3. 根据下面所讲的 Solver 方法，利用梯度更新参数；

4. 根据学习率（learning rate），历史数据和求解方法更新 solver 的状态,使权重从初始化状态逐步更新到最终的学习到的状态。solvers 的运行模式有 CPU/GPU 两种模式。

4.2 Methods

Solver 方法用于最小化损失（loss）值。给定一个数据集 D，优化的目标是 D 中所有数据损失的均值，即平均损失，取得最小值。

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

其中 $f_W(X^{(i)})$是数据中$X^{(i)}$项的损失，$r(W)$是正则项，权重为 λ。当 D 数据量很大时（译注：直接采用梯度下降方法计算量很大），在每一次迭代中，我们采用数据集的一个随机子集（mini-batch）来近似代替，其数据量远小于整个数据集（$N \ll |D|$）。

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

在前向过程（forward）中计算f_W（即 loss），在反向过程（backward）中计算∇f_W（即梯度 gradient）。根据误差梯度∇f_W、正则项的梯度$\nabla r(W)$ 以及其他方法的特定项来计算参数更新量ΔW。

4.2.1 SGD

随机梯度下降（Stochastic gradient descent, type:"SGD"）利用负梯度 $\nabla L(W)$和上一次权重的更新值V_t的线性组合来更新权重 W。学习率（learning rate）α 是负梯度的权重。动量（momentum）μ 是上一次更新值的权重。

有如下公式，根据上一次计算的更新值V_t和当前权重W_t来计算本次的更新值V_{t+1}和权重W_{t+1}：

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

$$W_{t+1} = W_t + V_{t+1}$$

学习的超参数（α 和 μ）需要一定的调整才能达到最好的效果。如果你不是很清楚该怎样做的话，请看下面的"经验法则（Rules of thumb）"，更多的细节请参考 Leon Bottou 的 Stochastic Gradient Descent Tricks[1]

[1] L. Bottou. Stochastic Gradient Descent Tricks. *Neural Networks: Tricks of the Trade: Springer, 2012.*

设定学习率 α 和动量 μ 的经验法则

一个比较好的建议是，将学习速率（ learning rate α）初始化为$\alpha \approx 0.01 = 10^2$，然后在训练（training）中当 loss 达到稳定时，将 α 除以一个常数（例如 10），将这个过程重复多次。对于动量（momentum μ）一般设置为$\mu = 0.9$，μ 使 weight 的更新更为平缓，使学习过程更为稳定、快速。

这是 Krizhevsky 在其著名的赢得 ILSVRC-2012 竞赛过程中使用的技巧 [1]，Caffe 中的 SolverParameter 让个技巧易于实现。

详见：`./examples/imagenet/alexnet_solver.prototxt.`

要使用上述技巧，可以将下面的代码添加到自定义的 solver prototxt 文件中：

```
base_lr: 0.01          # 开始学习速率为：α = 0.01=1e-2
lr_policy: "step"      # 学习策略: 每 stepsize 次迭代之后，将 α 乘以 gamma
gamma: 0.1             # 学习速率变化因子
stepsize: 100000       # 每 100K 次迭代，降低学习速率
max_iter: 350000       # 训练的最大迭代次数 350K
momentum: 0.9          # 动量 momentum 为：μ = 0.9
```

上面的例子中，我们将动量 μ 设为常数 0.9。刚开始的前 100K 次迭代时学习速率 α（*base_lr*）设置为$\alpha = 0.01 = 10^{-2}$，然后将学习速率 α 乘以 gamma(γ)，即在第 100K-200K 次迭代时以学习率为$\alpha' = \alpha\gamma = (0.01)(0.1) = 0.001 = 10^{-3}$进行训练，之后第 200K-300K 次迭代时学习速率为$\alpha'' = 10^{-4}$，最后第 301K-350K 次（因为设定的最大迭代次数为 350000）迭代时学习速率为$\alpha = 10^{-5}$。

上述例子中，当训练次数达到一定量后，更新值（update）会扩大到$\frac{\mu}{1-\mu}$倍，所以如果增加 μ 的话，最好是相应地减少 α 值（反之亦然）。

举个例子，设$\mu = 0.9$，则更新值会扩大$\frac{1}{1-0.9} = 10$倍。如果将 μ 扩大为 0.99，那么更新值会扩大 100 倍，所以 α 应该除以 10。

上述技巧也只是经验之谈，不保证绝对有用，甚至可能一点用也没有。如果训练过程中出现了发散现象（例如，loss 或者 output 值非常大导致显示 NAN，inf 这些符号），试着减小基准学习速率（例如 base_lr 设置为 0.001）再训练，重复这个过程，直到找到一个比较合适的学习速率(base_lr)。

[1] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet Classification with Deep Convolutional Neural Networks *Advances in Neural Information Processing Systems*, 2012.

4.2.2 AdaDelta

AdaDelta (type: "AdaDelta") 方法 (M. Zeiler [1]) 是一种“鲁棒的学习率方法”，同 SGD 一样是一种基于梯度的优化方法。更新方程如下：

$$
\begin{aligned}
(v_t)_i &= \frac{\mathrm{RMS}((v_{t-1})_i)}{\mathrm{RMS}(\nabla L(W)_t)_i}\left(\nabla L(W_{t'})\right)_i \\
\mathrm{RMS}\left(\nabla L(W_t)\right)_i &= \sqrt{E[g^2]+\varepsilon} \\
E[g^2]_t &= \delta E[g^2]_{t-1} + (1-\delta)g_t^2 \\
(W_{t+1})_i &= (W_t)_i - \alpha(v_i)_i
\end{aligned}
$$

[1] M. Zeiler ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. *arXiv preprint*, 2012.

4.2.3 AdaGrad

自适应梯度下降方法（ Adaptive gradient）[1]跟随机梯度下降（Stochastic gradient）一样是基于梯度的优化方法（作者文中比喻到：find needles in haystacks in the form of very predictive but rarely seen features）。给定之前更新的信息 $\left(\nabla L(W)\right)_{t'}$,对于

$t' \in \{1,2,\ldots,t\}$，权重 W 中每个元素 i 的更新如下：

$$
(W_{t+1})_i = (W_t)_i - \alpha\frac{(\nabla L(W))_i}{\sqrt{\Sigma_{t'=1}^{t}\left(\nabla L(W_{t'})\right)_i^2}}
$$

实际操作中，对于权重$W \in R^d$，自适应梯度（AdaGrad ）的实现只需要$O(d)$额外空间存储历史梯度信息（而不是$O(dt)$的存储空间来存储每一个单独的历史纪录）。

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

4.2.4 Adam

Adam 也是一种基于梯度的优化方法。它包含一对自适应时刻估计变量（ adaptive moment estimation）(m_t, μ_t)，可以看做是 AdaGrad 的一种泛化形式。

$$(\mathrm{m_t})_\mathrm{i} = \beta_1(m_{t-1})_i + (1-\beta_i)\big(\nabla L(W_t)\big)_i$$

$$(v_t)_i = \beta_2(v_{t-1})_i + (1-\beta_2)\big(\nabla L(W_t)\big)_i^2$$

$$(W_{t+1})_i = (W_t)_i - \alpha\frac{\sqrt{1-(\beta_2)_i^t}}{1-(\beta_1)_i^t}\ \frac{(m_t)_i}{\sqrt{(v_t)_i}+\varepsilon}$$

Kingma et al. [1]提出$\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$ 作为默认值，Caffe 中同样使用momemtum, momentum2, delta分别代表$\beta_1, \beta_2, \varepsilon$ 。

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

4.2.5 NAG

Nesterov 提出的加速梯度下降（Nesterov's accelerated gradient）是凸优化的一种最优算法 [1]，其收敛速度可以达到 $\mathrm{O}(1/\mathrm{t}^2)$，而不是$\mathrm{O}(1/\mathrm{t})$。尽管在使用 Caffe 训练深度神经网络时很难满足$\mathrm{O}(1/\mathrm{t}^2)$收敛条件（例如，由于非平滑 non-smoothness 、非凸 non-convexity），但实际中 NAG 对于某些特定结构的深度学习模型仍是一个非常有效的方法[2]。

权重 weight 更新参数与随机梯度下降（Stochastic gradient）非常相似：

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

$$W_{t+1} = W_t + V_{t+1}$$

与 SGD 的不同之处在于梯度$\nabla L(W)$项中取值不同：在 NAG 中， 我们取当前权重和动量之和的梯度$\nabla\mathrm{L}(\mathrm{W_t} + \mu\mathrm{V_t})$；在 SGD 中，只是简单的计算当前权重的动量$\nabla L(W_t)$。

[1] Y. Nesterov. A Method of Solving a Convex Programming Problem with Convergence Rate $\mathrm{O}(1/\sqrt{\mathrm{k}})$. Soviet Mathematics Doklady, 1983.

[2] 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.

4.2.6 RMSprop

Tieleman 在 Coursera 课程[1]中提到的 RMSprop 方法同样是一种基于梯度的优化方法（同 SGD 类似）。更新方程如下：

$$(v_t)_i = \begin{cases} (v_{t-1})_i + \delta, & \left(\nabla L(W_t)\right)_i \left(\nabla L(W_{t-1})\right)_i > 0 \\ (v_{t-1})_i \cdot (1-\delta), & else \end{cases}$$

$$(W_{t+1})_i = (W_t)_i - \alpha(v_t)_i$$

如果梯度更新值产生振动，则让梯度减小（ 乘以$1-\delta$），否则增加 δ。δ(rms_decay)默认值是 0.02。

[1] 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.

本章翻译、校对、审校人员：

翻译：王洪振

校对：巴君、它它

审校：辛淼

第五章 Layer Cataloge

为了创建一个 caffe 模型，我们需要在一个 protocol buffer(prototxt)文件中定义模型的结构。

在 caffe 中，层和相应的参数都定义在 caffe.proto 文件里。

5.1 视觉层 Vision Layers

头文件：./include/caffe/vision_layers.hpp

视觉层的输入与输出均为图像。一个典型的图像通常为单通道的灰度图或三通道的 RBG 彩色图。但本文所指图像是一个广义的概念，明显特性来自于空间结构:高和宽通常均大于 1 而通道数不限，类似结构的数据均可理解为图像。这种结构可以帮助 caffe 的层决定如何处理输入数据，具体来说，大多数视觉层通常是在输入数据的某块区域执行特定操作来产生对应的输出。相反的，其它类型的层通常会忽略空间结构而把输入图像看作是一个维度为 chw 的“单个大向量”。

5.1.1 卷积 Convolution

层类型 : Convolution

- CPU 实现代码 : ./src/caffe/layers/convolution_layer.cpp
- CUDA,GPU
- 实现代码 : ./src/caffe/layers/convolution_layer.cu
- 参数 (ConvolutionParameter convolution_param)
    - 必填项

      1. num_output (c_o): 指定卷积核的数量；

      2. kernel_size (或者 kernel_h 和 kernel_w): 指定卷积核的高度和宽度。
    - 强烈推荐

      weight_filler [default type: 'constant' value: 0]:

      （译者注：指定参数的初始化方案）
    - 可选项

1. bias_term [default true]: 指定是否给卷积输出添加偏置项；

2. pad (或者 pad_h 和 pad_w) [default 0]: 指定在输入图像周围补 0 的像素个数；

3. stride (或者 stride_h 和 stride_w) [default 1]: 指定卷积核在输入 图像上滑动的步长；

4. group (g) [default 1]: （译者注：指定分组卷积操作的组数，默认为 1 即不分组）如果 g >1,我们可以将卷积核的连接限制为输入数据的一个子集。具体地说，输入图像和输出图像在通道维度上分别被分成 g 个组，输出图像的第 i 组只与输入图像第 i 组连接（即输入图像的第 i 组与相应的卷积核卷积得到第 i 组输出）。

- 输入

  n * c_i * h_i * w_i

- 输出

  n * c_o * h_o * w_o， 式中 h_o = (h_i + 2 * pad_h - kernel_h) / stride_h + 1，w_o 计算方式类似。

- 实例 (见 ./models/bvlc_reference_caffenet/train_val.prototxt)

```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # 卷积核的局部学习率和权值衰减因子
  param { lr_mult: 1 decay_mult: 1 }
  # 偏置项的局部学习率和权值衰减因子
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
```

```
    num_output: 96     # 学习 96 组卷积核
    kernel_size: 11    # 卷积核大小为 11x11
    stride: 4          # 卷积核滑动步长为 4
    weight_filler {
      type: "gaussian" # 使用高斯分布随机初始化卷积核
      std: 0.01        # 高斯分布的标准差为 0.01 (默认均值：0)
    }
    bias_filler {
      type: "constant" # 使用常数 0 初始化偏置项 0
      value: 0}
   }
}
```

Convolution 层使用一系列可训练的卷积核对输入图像进行卷积操作，每组卷积核生成输出图像中的一个特征图。

5.1.2 池化 Pooling

- 层类型: Pooling
- CPU 实现代码: ./src/caffe/layers/pooling_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/pooling_layer.cu
- 参数 (PoolingParameter pooling_param)

  - 必填

    kernel_size(或者 kernel_h 和 kernel_w): 指定池化窗口的高度和宽度

  - 可选

    1. pool [default MAX]: 池化方法, 目前提供三种: 最大值池化, 均值池化,和 随机池化

    2. pad (或者 pad_h 和 pad_w) [default 0]: 指定在输入图像周围补 0 的像素个数

    3.stride (或者 stride_h 和 stride_w) [default 1]: 指定池化窗口在输入数据上滑动

的步长

- 输入

  n * c * h_i * w_i

- 输出

n * c * h_o * w_o, 式中 h_o 和 w_o 的计算方法与卷积层相同。

- 示例 (见 ./models/bvlc_reference_caffenet/train_val.prototxt)

```
layer {
  name: "pool1"
  type: "Pooling"
  bottom: "conv1"
  top: "pool1"
  pooling_param {
    pool: MAX
    kernel_size: 3 # 池化窗口大小为 3x3
    stride: 2      # 池化窗口在输入图像上滑动的步长为 2
  }
}
```

5.1.3 局部响应值归一化 Local Response Normalization (LRN)

- 层类型: LRN
- CPU 实现代码: ./src/caffe/layers/lrn_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/lrn_layer.cu
- 参数 (LRNParameter lrn_param)
  - 可选

    1. local_size [default 5]: 对于跨通道的归一化，该参数指参与求和的通道数，对于通道内的规范化，该参数指的是参与求和的方形区域的边长

    2. alpha [default 1]: 尺度参数（见下文）

    3. beta [default 5]: 指数参数（见下文）

4. norm_region [default ACROSS_CHANNELS]: 指定在通道之间进行规范化(ACROSS_CHANNELS)还是在通道内进行规范化(WITHIN_CHANNEL)。（译者注：在通道间指的是沿着通道维度操作，通道内指的是在特征图的二维平面内操作）

局部响应值归一化层通过对输入数据的局部归一操作执行了一种“侧抑制”的机制。 在 ACROSS_CHANNELS 模式下， 局部区域沿着临近通道延伸（译者注：而非在特征图的平面内），而没有空间扩展（即局部区域的形状为 local_size x 1 x 1）。在 WITHIN_CHANNEL 模式下， 局部区域在各自通道内部的图像平面上延伸（即局部区域的形状为 1 x local_size x local_size）。每个输入值除以 $(1+(\alpha/n)\sum_i x_i^2)^{\beta}$ 以实现归一化， 式中，n 是局部区域的大小，在以当前输入值为中心的区域内计算加和（如有需要，需在边缘补零）。

5.1.4 im2col

Im2col 是一个辅助操作， 用来实现图像到“列向量”的转换，通常情况下我们不需要了解其实现细节。Im2col 用在 CAFFE 早期卷积的矩阵乘法中，即将所有图像块组成一个矩阵。（译者注：在 caffe 中，执行卷积操作时，将图像中与卷积核作用的图像块写成列向量，然后将这些列向量按行的方向依次排开组成一个二维的矩阵，同时，每组卷积核写成一个行向量，多个输出通道对应的多组卷积核按列的方向依次排开形成一个二维矩阵，这样，卷积操作就转化为矩阵乘法操作。）

5.2 损失层 Loss Layers

Loss 设置了一个损失函数用来比较网络的输出和目标值，通过最小化损失来驱动网络的训练。网络的损失通过前向操作计算，网络参数相对于损失函数的梯度则通过反向操作计算。

5.2.1 Softmax 损失

- 层类型： SoftmaxWithLoss

softmax 损失层一般用于计算多类分类问题的损失，在概念上等同于 softmax 层后跟随一个多变量 Logistic 回归损失层(multinomial logistic loss)，但能提供数值上更稳定的梯度。

5.2.2 平方和/欧式损失 Sum-of-Squares / Euclidean

- 层类型: EuclideanLoss

Euclidean 损失层用来计算两个输入差值的平方和

$$\frac{1}{2N}\sum_{i=1}^{N}\left\|x_i^1 - x_i^2\right\|_2^2$$

5.2.3 Hinge / Margin 损失

- 层类型: HingeLoss
- CPU 实现代码: ./src/caffe/layers/hinge_loss_layer.cpp
- CUDA GPU 实现代码: 未实现
- 参数 (HingeLossParameter hinge_loss_param)
  - 可选
    - norm [default L1]: 正则项类型，目前有 L1 和 L2
- 输入
  - n * c * h * w 预测值
  - n * 1 * 1 * 1 真实标签
- 输出
  - 1 * 1 * 1 * 1 计算的损失
- 示例

```
 # L1 Norm
  layer {
  name: "loss"
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
}


# L2 Norm
  layer {
  name: "loss"
```

```
  type: "HingeLoss"
  bottom: "pred"
  bottom: "label"
  top: "loss"
  hinge_loss_param {
         norm: L2
         }
}
```

hinge 损失层用来计算 one-vs-all hinge 或者 squared hinge 损失。

5.2.3 交叉熵损失 Sigmoid Cross-Entropy

SigmoidCrossEntropyLoss

5.2.4 信息熵损失 Infogain

InfogainLoss

5.2.5 准确率 Accuracy and Top-k

Accuracy 用来计算网络输出相对目标值的准确率， 它实际上并不是一个损失层，所以没有反传操作。

5.3 激活层 Activation / Neuron Layers

一般来说，激活层执行逐个元素的操作， 输入一个底层 blob， 输出一个尺寸相同的顶层 blob。 在以下列出的这些层中，我们将忽略输入和输出 blob 的尺寸，因为它们是相同的:

- 输入 n * c * h * w
- 输出 n * c * h * w

5.3.1 ReLU / Rectified-Linear and Leaky-ReLU

- 层类型: ReLU
- CPU 实现代码: ./src/caffe/layers/relu_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/relu_layer.cu
- 参数 (ReLUParameter relu_param)
  - 可选

    negative_slope [default 0]: 设置激活函数在负数部分的斜率（默认为 0），

    对输入数据小于零的部分乘以这个因子，斜率为 0 时，小于零的部分完全滤掉。
- 示例 (见 ./models/bvlc_reference_caffenet/train_val.prototxt)

```
layer {
  name: "relu1"
  type: "ReLU"
  bottom: "conv1"
  top: "conv1"
}
```

给定一个输入数据， 当 x > 0 时， ReLU 层的输出为 x， 当 x <= 0 时， 输出为 negative_slope * x。 当 negative_slope 未指定时，等同于标准的 ReLU 函数 max(x, 0)，该层也支持原（址 in-place）计算， 即它的底层 blob 和顶层 blob 可以是同一个以节省内存开销。

5.3.2. Sigmoid

- 层类型: Sigmoid
- CPU 实现代码: ./src/caffe/layers/sigmoid_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/sigmoid_layer.cu
- 示例 (见 ./examples/mnist/mnist_autoencoder.prototxt)

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

```
}
```

The Sigmoid 层使用 sigmoid(x)函数计算每个输入数据的输出

5.3.3 TanH / Hyperbolic Tangent

- 层类型: TanH
- CPU 实现代码: ./src/caffe/layers/tanh_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/tanh_layer.cu
- 示例

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

The TanH 层使用 tanh(x)函数计算每个输入数据 x 的输出

5.3.4 Absolute Value

- 层类型: AbsVal
- CPU 实现代码: ./src/caffe/layers/absval_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/absval_layer.cu
- 示例

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

The AbsVal 层使用 abs(x)函数计算每个输入数据 x 的输出

5.3.5 Power

- 层类型: Power
- CPU 实现代码: ./src/caffe/layers/power_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/power_layer.cu
- 参数 (PowerParameter power_param)
  - 可选

    1.power [default 1]

    2.scale [default 1]

    3.shift [default 0]
- 示例

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

The Power 层使用函数(shift + scale * x) ^ power 计算每个输入数据 x 的输出

5.3.5 BNLL

- 层类型: BNLL
- CPU 实现代码: ./src/caffe/layers/bnll_layer.cpp

- CUDA GPU 实现代码: ./src/caffe/layers/bnll_layer.cu
- 示例

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

The BNLL 层使用函数 log(1 + exp(x))计算每个输入数据 x 的输出。

5.4 数据层 Data Layers

数据能过数据层进入 caffe 网络：数据层处于网络的最底层， 数据可以从高效率的数据库中读取（如 LevelDB 或 LMDB）， 可以直接从内存中读取， 若对读写效率要求不高也可以从硬盘上的 HDFT 文件或者普通的图片文件读取。

常见的数据预处理操作（减均值，尺度变换，随机裁剪或者镜像）可以能过设定参数 TransformationParameter 来实现。

5.4.1 数据库 Database

- 层类型: Data
- 参数
  - 必填

    1. source: 数据库文件的路径

    2. batch_size: 网络单次输入数据的数量
  - 可选

    1. rand_skip: 跳过开头的 rand_skip * rand(0,1)个数据，通常在异步随机梯度下降法里使用；

    2. backend [default LEVELDB]: 选择使用 LEVELDB 还是 LMDB。

5.4.2 内存数据 In-Memory

- 层类型: MemoryData
- 参数
  - 必填

batch_size, channels, height, width: 指定从内存中读取的输入数据块的尺寸

memory data 层直接从内存中读取数据而不用拷贝。使用这个层时需要调用 MemoryDataLayer::Reset (C++) 或者 Net.set_input_arrays(Python) 来指定数据来源（四维按行存储的数组）， 每次读取一个大小为 batch-sized 的数据块。

5.4.3 HDF5 Input

- 层类型: HDF5Data
- 参数
  - 必填

1. source: 文件路径；
2. batch_size。

5.4.4 HDF5 Output

- 层类型: HDF5Output
- 参数
  - 必填

file_name: 写入文件的路径

HDF5 output 层执行了一个与数据读取相反的操作， 它将数据写进硬盘。

5.4.5 图像数据 Images

- 层类型: ImageData

- 参数
  - 必填

1. source: text 文件的路径名，该 text 文件的每一行存储一张图片的路径名和对应的标签；

2. batch_size: 打包成 batch 的图片数量。

  - 可选

1. rand_skip

2. shuffle [default false]

3. new_height, new_width: 根据设置的值，输入的图片将会被调整成给定的尺寸。

5.4.6 窗口 Windows

WindowData

5.4.7 Dummy

DummyData 主要用来开发和调试， 详细介绍见 DummyDataParameter。

5.5 普通层 Common Layers

5.5.1 内积 / 全连接 Inner Product

- 层类型: InnerProduct
- CPU 实现码: ./src/caffe/layers/inner_product_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/inner_product_layer.cu
- 参数 (InnerProductParameter inner_product_param)
  - 必填

num_output (c_o): 层的输出节点数或者理解为滤波器的个数

  - 推荐

weight_filler [default type: 'constant' value: 0]

  - 可选

1.bias_filler [default type: 'constant' value: 0]

2.bias_term [default true]: 指定是否给每个滤波器添加并训练偏置项

- 输入

  n * c_i * h_i * w_i

- 输出

  n * c_o * 1 * 1

- 示例

```
layer {
 name: "fc8"
 type: "InnerProduct"
 # learning rate and decay multipliers for the weights
 param { lr_mult: 1 decay_mult: 1 }
 # learning rate and decay multipliers for the biases
 param { lr_mult: 2 decay_mult: 0 }
 inner_product_param {
   num_output: 1000
   weight_filler {
     type: "gaussian"
     std: 0.01
   }
   bias_filler {
     type: "constant"
     value: 0
   }
 }
 bottom: "fc7"
 top: "fc8"
}
```

InnerProduct 层（也被称作全连接层）将输入看成一个一向量，输出也为向量（输出 blob 的高和宽都为 1）。

5.5.2 分裂 Splitting

The Split 是一个可以将输入的 blob 分裂（复制）成多个输出 blob 的功能层， 通常当一个 blob 需要给多个层作输入数据时该层会被使用。

5.5.3 摊平 Flattening

latten 层用来将尺寸为 n * c * h * w 的输入 blob 转换成一个尺寸为 n * (c*h*w) 的输出 blob。

5.5.4 变形 Reshape

- 层类型: Reshape
- 实现代码: ./src/caffe/layers/reshape_layer.cpp
- 参数 (ReshapeParameter reshape_param)

  可选：（见下文）

  shape

  ○ 输入

  任意维度的 blob

  ○ 输出

  按照参数 reshape_param 修改维度的 blob
- 示例

```
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
    }
  }
}
```

Reshape 层在不改变数据的情况下改变输入 blob 的维度，和 Flatten 操作一样，处理过程只在输入 blob 上进行，没有进行数据的拷贝。

输出数据的维度通过参数 ReshapeParam 设定， 可以使用正数直接指定输出 blob 的相应维度， 也可以使用两个特殊的值来设定维度：

- 0 表示从层的底层 blob 中直接取相应的维度，即不改变对应的维度。 例如，在参数中设置第一个维度为 dim: 0，底层 blob 在第一个维度上是 2，则输出的顶层 blob 的第一个维度也是 2
- -1 表示用其它的维度计算该维度的值。这个操作与 numpy 中的－1 或者 MATLAB 中 reshap 操作时的[]相似：对应的维度通过保证 blob 中数据的总个数不变来计算， 因此，在 reshape 操作中最多设定一个－1

从另外一个例子可以看出， 通过设定参数 reshape_param { shape { dim: 0 dim: -1 } }我们可以得到与 Flatten 层相同的操作结果。

5.5.5 连结 Concatenation

- 层类型: Concat
- CPU 实现代码: ./src/caffe/layers/concat_layer.cpp
- CUDA GPU 实现代码: ./src/caffe/layers/concat_layer.cu
- 参数 (ConcatParameter concat_param)
  - 可选

axis [default 1]: 0 表示沿着样本个数的维度(num)串联， 1 表示沿着通道维度(channels)串联。

- 输入

  n_i * c_i * h * w 对于第 i 个输入 blob， i 的取值为{1,2,...,K}

- 输出

  1. if axis = 0: (n_1 + n_2 + ... + n_K) * c_1 * h * w， 所有输入 blob 在通道上的维度 c_i 需相同；

  2. if axis = 1: n_1 * (c_1 + c_2 + ... + c_K) * h * w， 所有输入 blob 在样本个数上的维度 n_i 需相同。

- 示例

```
  layer {
  name: "concat"
  bottom: "in1"
  bottom: "in2"
  top: "out"
  type: "Concat"
  concat_param {
    axis: 1
  }
}
```

Concat 层用来将多个输入 blob 连结成一个 blob 输出

5.5.6 切片 Slicing

Slice 层按照给定维度（num 或者 channel）和切分位置的索引将一个输入 blob 分成多个 blob 输出。

- 示例

```
  layer {
  name: "slicer_label"
```

```
  type: "Slice"
  bottom: "label"
  ## Example of label with a shape N x 3 x 1 x 1
  top: "label1"
  top: "label2"
  top: "label3"
  slice_param {
    axis: 1
    slice_point: 1
    slice_point: 2
  }
}
```

axis 指定执行切分操作的所在的维度；slice_point 指定所选维度上切分位置的索引（索引的个数需等于顶层 blob 的个数减 1）。

5.5.7 逐个元素操作 Elementwise Operations

Eltwise

5.5.8 Argmax

ArgMax

5.5.9 Softmax

Softmax

5.5.10 Mean-Variance Normalization

MVN

本章翻译、校对、审校人员：

翻译：孙琳钧

校对：占鸿渐、李文杰

审校：刘昕

第六章 Interfaces

Caffe 有命令行、Python 和 MATLAB 三种接口，来实现日常使用、研究代码的交互以及实现快速原型。Caffe 以 C++库为核心，其在开发中使用模块化接口，而不是每次都调用其定义的编译。cmdcaffe，pycaffe 与 matcaffe 接口都可供用户使用。

6.1 Command Line

命令行接口 - cmdcaffe - 是 caffe 中用来模型训练，计算得分以及方法判断的工具。没有附加参数的情况下运行 caffe 可得到帮助提示。caffe 与其它工具存放在 caffe/build/tools 目录下。（这些例子要求你先要完成 LeNet / MNIST 的例子）。

6.1.2 训练

caffe train 命令可以从零开始学习模型，也可以从已保存的 snapshots 继续学习，或将已经训练好的模型应用在新的数据与任务上进行微调即 fine-tuning 学习：

- 所有的训练都需要添加-solver solver.prototxt 参数完成 solver 的配置。
- 继续训练需要添加-snapshot model_iter_1000.solverstate 参数来加载 solver snapshot。
- Fine-tuning 需要添加-weights model.caffemodel 参数完成模型初始化。

例如，可以运行如下代码：

```
# 训练 LeNet
caffe train -solver examples/mnist/lenet_solver.prototxt
# 在 2 号 GPU 上训练
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 2
# 从中断点的 snapshot 继续训练
caffe train -solver examples/mnist/lenet_solver.prototxt -snapshot
examples/mnist/lenet_iter_5000.solverstate
```

对于 fine-tuning 的完整例子，可以参考 examples/finetuning_on_flickr_style，但是只调用训练命令如下：

```
# 微调 CaffeNet 模型的权值以完成风格识别任务（style recognition）
caffe train -solver examples/finetuning_on_flickr_style/solver.prototxt
-weights models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel
```

6.1.2 测试

caffe test 命令通过在 test phase 中运行模型得到分数，并且用这分数表示网络输出的最终结果。网络结构必须被适当定义，生成 accuracy 或 loss 作为其结果。测试过程中，终端会显示每个 batch 的得分，最后输出全部 batch 得分的平均值。

```
# 对于网络结构文件 lenet_train_test.prototxt 所定义的网络
# 用 validation set 得到已训练的 LeNet 模型的分数
caffe test -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 100
```

6.1.3 Benchmarking

caffe time 命令 通过逐层计时与同步，执行模型检测。这是用来检测系统性能与测量模型相对执行时间。

```
# （这些例子要求你先要完成 LeNet / MNIST 的例子）
# 在 CPU 上，10 iterations 训练 LeNet 的时间
caffe time -model examples/mnist/lenet_train_test.prototxt -iterations 10
# 在 GPU 上，默认的 50 iterations 训练 LeNet 的时间
caffe time -model examples/mnist/lenet_train_test.prototxt -gpu 0
# 在第一块 GPU 上，10 iterations 训练已给定权值的网络结构的时间
caffe time -model examples/mnist/lenet_train_test.prototxt -weights
examples/mnist/lenet_iter_10000.caffemodel -gpu 0 -iterations 10
```

6.1.4 诊断

caffe device_query 命令对于多 GPU 机器上，在指定的 GPU 运行，输出 GPU 细节信息用

来参考与检测设备序号。

```
# 查询第一块 GPU
caffe device_query -gpu 0
```

6.1.5 并行模式

caffe 工具的 -gpu 标识，对于多 GPU 的模式下，允许使用逗号分开不同 GPU 的 ID 号。solver 与 net 在每个 GPU 上都会实例化，因此 batch size 由于具有多个 GPU 而成倍增加，增加的倍数为使用的 GPU 块数。如果要重现单个 GPU 的训练，可以在网络定义中适当减小 batch size。

```
# 在序号为 0 和 1 的 GPU 上训练 （ 双倍的 batch size ）
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu 0,1
# 在所有 GPU 上训练 （ 将 batch size 乘上 GPU 数量)
caffe train -solver examples/mnist/lenet_solver.prototxt -gpu all
```

6.2 Python

Python 接口 – pycaffe – 是 caffe 的一个模块，其脚本保存在 caffe/python。通过 import caffe 加载模型，实现 forward 与 backward、 IO、网络可视化以及求解模型等操作。所有的模型数据，导数与参数都可读取与写入。

- caffe.Net 是加载、配置和运行模型的中心接口
- caffe.Classsifier 与 caffe.Detector 为一般任务实现了便捷的接口
- caffe.SGDSolver 表示求解接口
- caffe.io 通过预处理与 protocol buffers，处理输入/输出
- caffe.draw 实现数据结构可视化
- Caffe blobs 通过 numpy ndarrays 实现易用性与有效性

IPython notebooks 教程在 caffe/examples: 可通过 ipython notebook caffe/examples 来尝试使用。对于开发者的参考提示贯穿了整个代码。

make pycaffe 可编译 pycaffe。通过 export PYTHONPATH= /path/to/caffe/python: $PYTHONPATH 将模块目录添加到自己的 $PYTHONPATH 目录，或者相似的指令来实现 import caffe。

6.3 MATLAB

MATLAB 接口(matcaffe)是在 caffe/matlab 路径中的 caffe 软件包。在 matcaffe 的基础上，可将 Caffe 整合进你的 Matlab 代码中。

在 MatCaffe 中，你可实现：

- 在 Matlab 中创建多个网络结构(nets)
- 进行网络的前向传播(forward)与反向传导(backward)计算
- 存取网络中的任意一层，或者网络层中的任意参数
- 在网络中，读取数据或误差，将数据或误差写入任意层的数据(blob)，而不是局限在输入 blob 或输出 blob
- 保存网络参数进文件，从文件中加载
- 调整 blob 与 network 的形状
- 编辑网络参数，调整网络
- 在 Matlab 中，创建多个 solvers 进行训练
- 从 solver 快照(snapshots)恢复并继续训练
- 在 solver 中，访问训练网络(train nets)与测试网络(test nets)
- 迭代一定次数后将网络结构交回 Matlab 控制
- 将梯度方法融合进任意的 Matlab 代码中

caffe/matlab/demo/classification_demo.m 中有一个 ILSVRC 图片的分类 demo（需要在 Model Zoo 中下载 BVLC CaffeNet 模型）。

6.3.1 编译 MatCaffe

用 make all matcaffe 命令编译 MatCaffe。随后，需要用 make mattest 命令进行测试。

经常遇到的问题：如果在 make mattest 过程中，得到错误信息，例如 libstdc++.so.6:version 'GLIBCXX_3.4.15' not found，通常表示你 Matlab 的运行时刻库(runtime library)与你的编译时刻库(compile-time library)不匹配。需要在启动 Matlab 之前，完成如下操作：

```
export LD_LIBRARY_PATH=/opt/intel/mkl/lib/intel64:/usr/local/cuda/lib64
export LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libstdc++.so.6
```

或者根据你的系统更改位置，再次运行 make mattest 确认问题是否解决。注意：这个问

题有时候更复杂，因为启动 Matlab 可能会覆盖你的 LD_LIBRARY_PATH 环境变量。

你可以在 Matlab 中运行!ldd ./matlab/+caffe/private/caffe_.mexa64（mex 拓展名可能在你的系统上是不同的）查来运行时刻库，然后通过将其增加到你的 LD_PRELOAD 环境变量来预加载编译时刻库。

在构建和测试成功后，将这个包添加进 Matlab 搜索路径：在 caffe 根目录启动 matlab，并在 Matlab 命令窗口运行以下命令：

```
addpath ./matlab
```

你可以在 Matlab 中运行 savepath 来保存 Matlab 搜索路径。这样你就不需要在每次运行 MatCaffe 时再重新运行一遍上述指令。

6.3.2 使用 MatCaffe

MatCaffe 在使用上与 PyCaffe 相似。

下面的例子展示了具体用法，假设你已在 Model Zoo 中下载了 BVLC CaffeNet 并从 caffe 根目录中启动了 matlab。

```
model = './models/bvlc_reference_caffenet/deploy.prototxt';
weights
='./models/bvlc_reference_caffenet/bvlc_reference_caffenet.caffemodel';
```

6.3.2.1 设置运行模式和 GPU 设备

在创建 net 或 solver 之前，应该先设置运行模式以及 GPU 设备。

使用 CPU：

```
caffe.set_mode_cpu();
```

使用 GPU，指定 GPU 的 ID 号

```
caffe.set_mode_gpu();
caffe.set_device(gpu_id);
```

6.3.2.2 创建网络，保存及读取其 layers 和 blobs

创建网络：

```
net = caffe.Net(model, weights, 'test'); % 创建网络并加载权值
```

或者

```
net = caffe.Net(model, 'test'); % 创建网络，但不加载权值
net.copy_from(weights); % 加载权值
```

创建的 net 对象为

```
  Net with properties:

        layer_vec: [1x23 caffe.Layer]
         blob_vec: [1x15 caffe.Blob]
           inputs: {'data'}
          outputs: {'prob'}
  name2layer_index: [23x1 containers.Map]
   name2blob_index: [15x1 containers.Map]
      layer_names: {23x1 cell}
           blob_names: {15x1 cell}
```

两个 containers.Map 对象用于找到 layer 或 blob 的标号(name)。

你可以存取网络中任意的 blob。若希望用 1 来填充’data’这个 blob：

```
net.blobs('data').set_data(ones(net.blobs('data').shape));
```

若希望将 ‘data’这个 blob 中的所有值都乘以 10：

```
net.blobs('data').set_data(net.blobs('data').get_data() * 10);
```

因为 Matlab 的标号从 1 开始，且以列(column)为主，则 blob 通常的 4 个维度在 Matlab

中用 [width, height, channels, num]表示，width 是第一维。另外，图像通道形式为 BGR。Caffe 使用单精度 float 型数据。如果你的数据不是单精度的，set_data 会自动将数据转换为单精度。

你可以访问每一层，对网络进行调整(surgery)。例如，将 conv1 的参数乘以 10：

```
net.params('conv1', 1).set_data(net.params('conv1', 1).get_data() * 10); % 设置权值
net.params('conv1', 2).set_data(net.params('conv1', 2).get_data() * 10); % 设置偏置
```

或者，你也可以使用以下命令：

```
net.layers('conv1').params(1).set_data(net.layers('conv1').params(1).get_data() * 10);
net.layers('conv1').params(2).set_data(net.layers('conv1').params(2).get_data() * 10);
```

保存刚刚修改的网络：

```
net.save('my_net.caffemodel');
```

得到 layer 的类型（返回一个 string）：

```
layer_type = net.layers('conv1').type;
```

6.3.2.3 网络的前向传导(forward)与后向传播(backward)

Forward 可使用 net.forward 或者 net.forward_prefilled 函数。net.forward 传入包含了输入数据的 N-D arrays 形式的单元阵列（cell array），返回包含输出数据的单元阵列。函数 net.forward_prefilled 使用了 forward 过程中，input blobs(s)中已经存在的数据。如果没有输

入就不会产生输出。在创建了类似 data = rand(net.blobs('data').shape);的输入数据后，可运行

```
res = net.forward({data});
prob = res{1};
```

或者

```
net.blobs('data').set_data(data);
net.forward_prefilled();
prob = net.blobs('prob').get_data();
```

Backward 用法相似，使用 net.backward 或者 net.backward_prefilled，用 get_diff 和 set_diff 替换 get_data 和 set_data。在创建了类似 prob_diff = rand(net.blobs('prob').shape); 的输出 blobs 梯度之后，你可以运行

```
res = net.backward({prob_diff});
data_diff = res{1};
```

或者

```
net.blobs('prob').set_diff(prob_diff);
net.backward_prefilled();
data_diff = net.blobs('data').get_diff();
```

然而，上述的 backward 计算不能得到正确结果，因为 Caffe 认为它不需要 backward 计算。为了得到正确的 backward 结果，你需要在网络 prototxt 文件中设置'force_backward: true' 。

在执行完 forward 或 backward，可在内部 blobs 得到数据或误差。例如，在 forward 之后提取 pool5 特征：

```
pool5_feat = net.blobs('pool5').get_data();
```

6.3.2.4 调整网络形状

假设一次性只在神经网络中放入一张（而不是十张）图像：

```
net.blobs('data').reshape([227 227 3 1]); % 改造 blob 'data'
```

```
net.reshape();
```

这样，整个网络形状就会被调整。现在 net.blobs('prob').shape 应该为[1000 1];

6.3.2.5 训练网络

假设已经创建了 ImageNET Tutorial 的训练和验证的 lmdb 数据库。若要创建在 LSVRC 2012 分类数据集上的 Solver 并进行训练：

```
solver = caffe.Solver('./models/bvlc_reference_caffenet/solver.prototxt');
```

创建的 solver 对象为

```
    Solver with properties:
          net: [1x1 caffe.Net]
    test_nets: [1x1 caffe.Net]
```

训练时，使用以下命令：

solver.solve();

或者只训练 1000 iterations（以便在训练更多的 iterations 之前，你可以对网络做一些其它修改）

```
solver.step(1000);
```

得到迭代次数：

```
iter = solver.iter();
```

得到训练/测试网络：

```
train_net = solver.net;
test_net = solver.test_nets(1);
```

从“your_snapshot.solverstate”的 snapshot 继续训练：

```
solver.restore('your_snapshot.solverstate');
```

6.3.2.6 Input and output

caffe.io 类提供了基础的输入函数 load_image 和 read_mean。例如，读取 ILSVRC 2012 的均值文件（假设你已经运行了./data/ilsvrc12/get_ilsvrc_aux.sh 下载与 imagenet 有关的文件）：

```
mean_data = caffe.io.read_mean('./data/ilsvrc12/imagenet_mean.binaryproto');
```

为了读取 Caffe 例子的图片，重新调整尺寸为[width, height]，假设我们想要 width = 256; height = 256;

```
im_data = caffe.io.load_image('./examples/images/cat.jpg');
im_data = imresize(im_data, [width, height]); % resize using Matlab's imresize
```

记住 width 是第一维度，通道为 BGR，这与 Matlab 通常存储图片的方式不同。 如果你不想使用 caffe.io.load_image，而想使用 Matlab 自带接口加载图像，你可以这样做：

```
im_data = imread('./examples/images/cat.jpg'); % read image
im_data = im_data(:, :, [3, 2, 1]); % 从 RGB 转换为 BGR
im_data = permute(im_data, [2, 1, 3]); % 改变 width 与 height 位置
im_data = single(im_data); % 转换为单精度
```

另外，你可以看一下 caffe/matlab/demo/classification_demo.m，理解怎么通过剪裁(crop)一张图像得到输入。在 caffe/matlab/hdf5creation 中展示了如何用 Matlab 读取与写入 HDF5 数据。因为 Matlab 自身在输出方面的功能十分强大，所以我们不提供额外的函数实现数据的输出。

6.3.2.7 清除 Nets 和 Solvers

Call caffe.reset_all() 清除你创建的所有 solvers 与/或独立的 nets。

本章翻译、校对、审校人员：

翻译：戴佳伦

校对：吴尚轩、刘艳飞

审校：辛淼

第七章 数据

7.1 数据：输入与输出

Caffe 中数据流以 Blobs 进行传输。数据层将输入转换为 blob 加载数据，将 blob 转换为其他格式保存输出。均值消去、特征缩放等基本数据处理都在数据层进行配置。新的数据格式输入需要定义新的数据层，网络的其余部分遵循 Caffe 中层目录的模块结构设定。

数据层的定义：

```
layer {
  name: "mnist"
  # 数据层加载 leveldb 或 lmdb 的数据库存储格式保证快速传输
  type: "Data"
  # 第一个顶部（top）是数据本身：“data”的命名只是方便使用
  top: "data"
  # 第二个顶部（top）是数据标签：“label”的命名只是方便使用
  top: "label"
  # 数据层具体配置
  data_param {
    # 数据库路径
    source: "examples/mnist/mnist_train_lmdb"
    # 数据库类型：LEVELDB 或 LMDB（LMDB 支持并行读取）
    backend: LMDB
    # 批量处理，提高效率
    batch_size: 64
  }
  # 常用数据转换
  transform_param {
    # 特征归一化系数，将范围为[0, 255]的 MNIST 数据归一化为[0, 1]
    scale: 0.00390625
  }
```

```
}
```

加载 MNIST 数字数据。

顶部和底部（Top and Bottom）： 数据层从 top 的 blobs 向模型输出数据，但没有 bottom 的 blobs，因为数据层没有输入。

数据与标签（Data and Label）：数据层至少要有一个 top 输出，规范化地命名为 **data**，对于 ground truth，第二个 top 输出，规范化地命名为 **label**。这两个 top 只是简单地生成 blobs，它们的名称没有特殊含义。（data, label）映射关系对于分类模型更方便。

转换（Transformations）：在数据层的定义中，数据预处理通过转换参数来定义。

```
layer {
  name: "data"
  type: "Data"
  [...]
  transform_param {
    scale: 0.1
    mean_file_size: mean.binaryproto
    # 对 images 进行水平镜像处理或者随机裁剪处理
    # 可作为简单的数据增强处理
    mirror: 1  # 1 = on, 0 = off
    # 裁剪块大小为 `crop_size` x `crop_size`:
    # - 训练时随机处理
    # - 测试时从中间开始
 crop_size: 227
  }
}
```

预获取（Prefetching）：为了提高网络吞吐量，数据层在网络计算当前数据块的同时在后台获取并准备下一个数据块。

多个输入（Multiple Inputs）：网络可以有任意数量和类型的输入。可根据需要定义任意数量的数据层，只要保证它们有唯一的 name 和 top。多输入对于非常见形式的 ground truth

是十分有用的。例如一个数据层读取真实数据，另一个数据层读取 ground truth。在这种设计下，data 和 label 可以是任意的 4D 数组。多输入的更进一步的应用是多模型和序列模型。在这些情况下，您可能需要实现自己的数据准备程序或者构建一个特殊的数据层。

欢迎提供更多格式与辅助工具，用来提升数据预处理。

7.2 格式

参照数据层的层目录，可以查看 Caffe 中数据格式的具体细节。

7.3 部署输入

对于“动态计算部署”（on-the-fly computation deployment）环境中，网络在输入域中定义其输入：这些网络直接接收数据来进行在线或交互式计算。

本章翻译、校对、审校人员：

翻译：杨轶斐

校对：戴嘉伦

审校：辛淼