

深度学习入门软件 使用手册



zhongtaovip@163.com

2016 年 6 月 27 日



一、目的

深度学习是一类新兴的多层神经网络学习算法，因其缓解了传统训练算法的最小性，引起机器学习领域的广泛关注。很多同学都希望了解和学习深度学习，让深度学习成为研究工作的一项工具。为了能让初学者更快的学会掌握深度学习的使用方法和强大之处，本文主要工作是在 Ubuntu 上使用 QT 平台，基于开源库 Caffe (<https://github.com/BVLC/caffe>)编写一个操作简便，功能较为完备的学习软件。

二、背景

深度学习的概念最早由多伦多大学的 G. E. Hinton 等于 2006 年提出，指基于样本数据通过一定的训练方法得到包含多个层级的深度网络结构的机器学习过程。传统的神经网络随机初始化网络中的权值，导致网络很容易收敛到局部最小值，为解决这一问题，Hinton 提出使用无监督预训练方法优化网络权值的初值，再进行权值微调的方法，拉开了深度学习的序幕。

而深度学习算法中较成熟和广泛使用的是卷积神经网络(CNN)，卷积神经网络是人工智能网络的一种，已成为当前语音分析和图像识别研究领域的热点，由于该网络避免了对图像的复杂前期预处理，可以直接输入原始图像，因而得到了更为广泛的应用。

卷积网络是为识别二维形状而特殊设计的一个多层感知器，这种网络结构对平移、比例缩放、倾斜或者其他形式的变形具有高度不变性。这些良好的性能是网络在有监督方式下学会的，网络的结构包括如下形式的约束：

(1) 特征提取：每个神经元的输入与前一层的局部接受域相连，并提取该局部的特征。一旦该局部特征被提取后，它与其它特征间的位置关系也随之确定下来。

(2) 特征映射：网络的每个计算层由多个特征映射组成，每个特征映射是一个平面，平面上所有神经元的权值相等。特征映射结构采用影响函数核小的 sigmoid 函数作为卷积网络的激活函数，使得特征映射具有位移不变性。

(3) 子抽样：每个卷积层跟着一个实现局部平均和子抽样的计算层，由此特征映射的分辨率降低。这种操作具有使特征映射的输出对平移和其他形式的变形的敏感度下降的作用

Caffe 是一个清晰而高效的深度学习框架，其作者是博士毕业于 UC Berkeley 的贾扬清。caffe 的好处是，我们基本上可以用一个比较简单的语言 (google protobuf) 来定义许多网络结构，然后我们可以在 CPU 或者 GPU 上面执行这些代码，而且 cpu 和 gpu 在数学结果上是兼容的。然后，所有的模型和网络都会公布出来，使得我们可以很容易地对互相发布的结果借鉴和参考。

Qt 是一个跨平台的 C++图形用户界面应用程序框架。它为应用程序开发者提供建立艺术级图形用户界面所需的所有功能。它是完全面向对象的，很容易扩展，并且允许真正的组件编程。Qt 是诺基亚公司的一个产品。1996 年，Qt 进入商业领域，已成为全世界范围内数千种成功的应用程序的基础。它也是目前流行的 Linux 桌面环境 KDE 的基础。结合 Qt 的优点，本文最终选择在 Qt 上开发此程序。



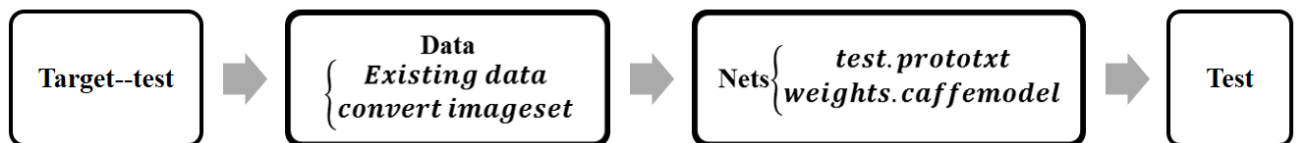
三、主要功能

一个完整的深度学习实验过程分为训练和测试两部分，在 `caffe` 框架下主要实现过程如下：

④ Train:



④ Test:



本文编写的程序界面如下图所示：

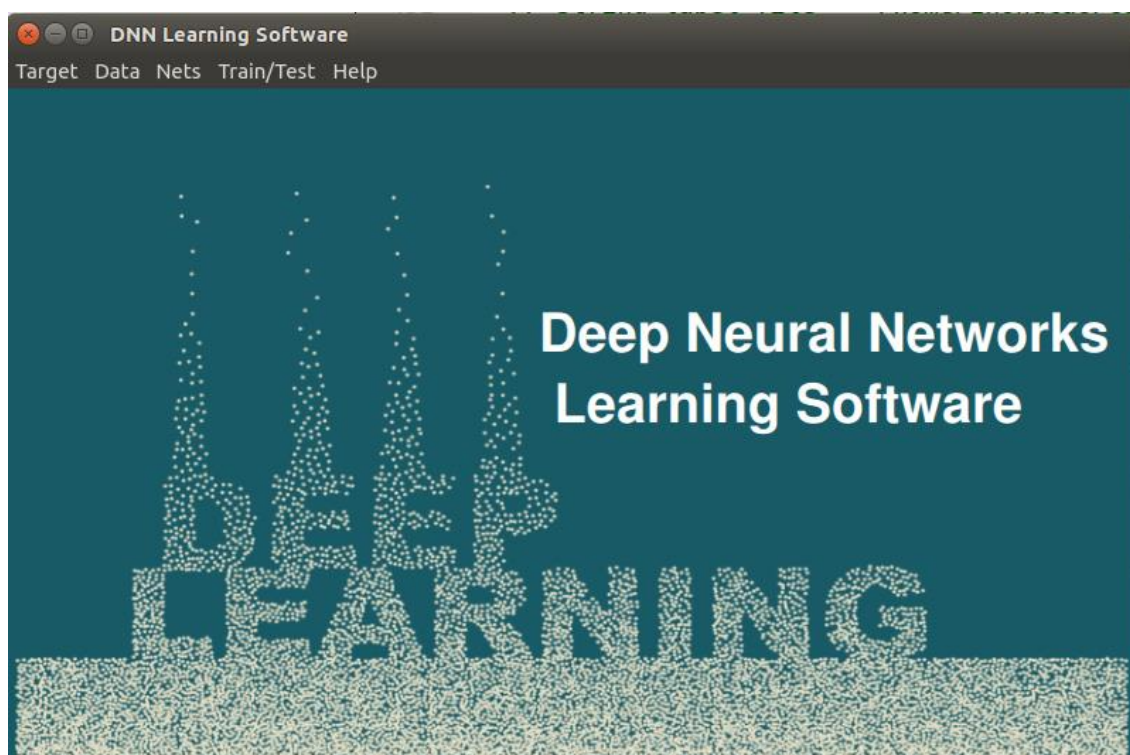


图 1 程序开始界面

1.Target 功能

程序中此功能主要作用是用户在使用时，先选择此次使用的目的，可以防止在后面的操作中有误操作。具体来说当选择 **train** 功能时，对应 **test** 的那些操作是不可用的。实现结果如下：

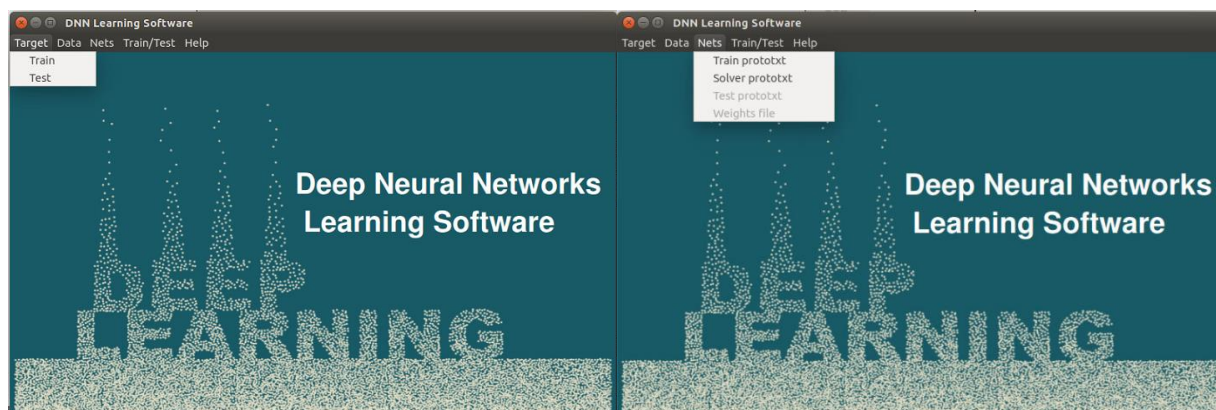


图 2 a)选择 train 功能

b)防止误操作

同理，在选择 **test** 功能时，对应的 **train** 的操作也不可选。

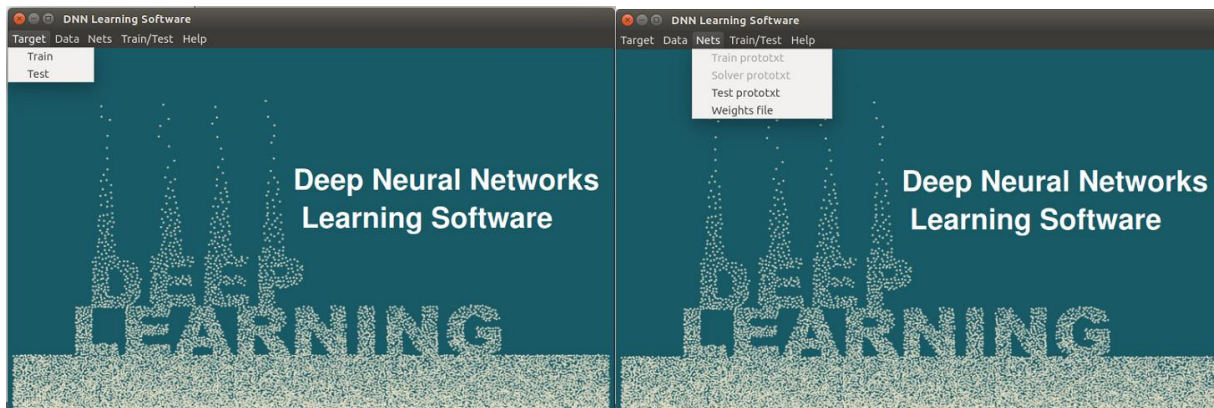


图 3 a)选择 test 功能

b)防止误操作

2.Data

数据是训练和测试的基础，在 `caffe` 框架中，`caffe` 支持 `lmdb/leveldb`、`hdf5` 等格式，而对于常见的图片格式，所以需要有一个数据转换功能。根据实际需要，可以将图片转换为 `lmdb` 格式，并可以将图片 `resize` 成所需要的大小，以针对不同的网络需求。

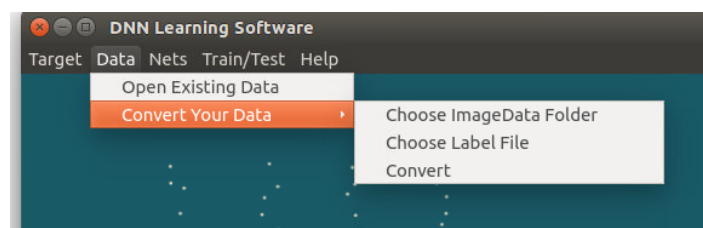


图 4 数据选择功能

对于需要转换的图片数据集，通过菜单栏对就的按钮，选择图片的根目录和标签文件，将会跳出窗口显示关于 `resize` 的设置及转换好的数据集保存位置。

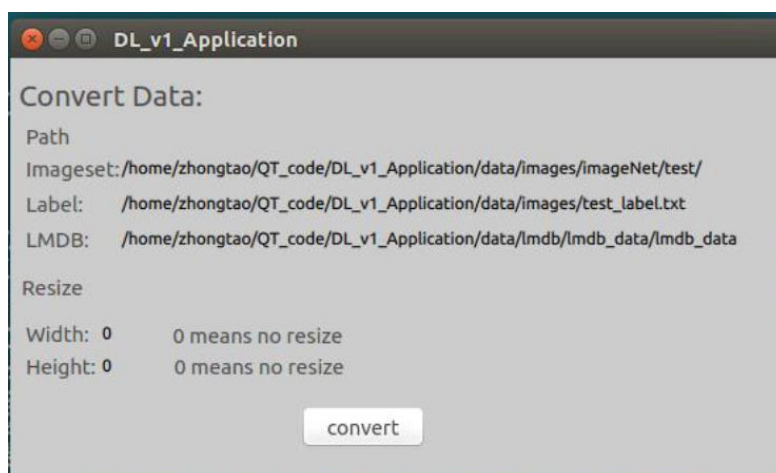




图 5 转换数据窗口

3.Nets

Caffe 框架中，对一个深度网络的定义都在根据 google protobuf 协议写的配置文件中，而对于训练和测试，所需要选择的配置文件是不同的，对于训练，主要是 train.prototxt 和 solver.prototxt 文件。

Train.prototxt 文件是描述整个网络的构造。比如 CNN 中最常见的一个卷积层，其实际描述规则如下：

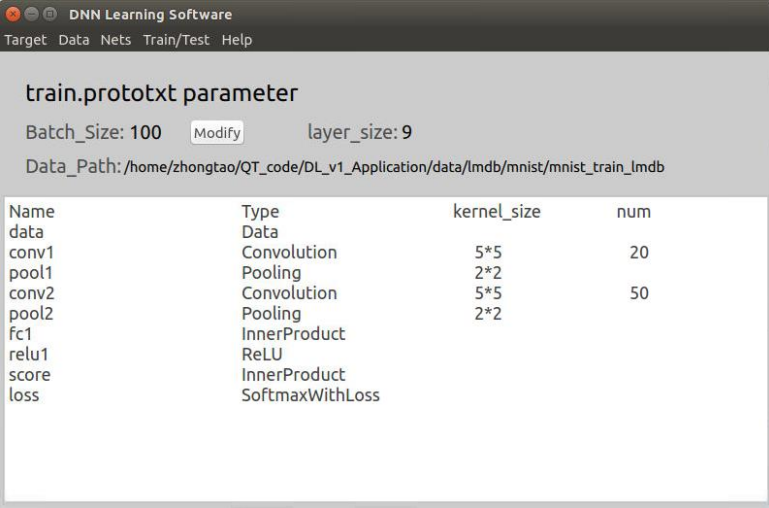
```
layer {
  name: "conv1"
  type: "Convolution"
  bottom: "data"
  top: "conv1"
  # learning rate and decay multipliers for the filters
  param { lr_mult: 1 decay_mult: 1 }
  # learning rate and decay multipliers for the biases
  param { lr_mult: 2 decay_mult: 0 }
  convolution_param {
    num_output: 96      # learn 96 filters
    kernel_size: 11     # each filter is 11x11
    stride: 4           # step 4 pixels between each filter application
    weight_filler {
      type: "gaussian" # initialize the filters from a Gaussian
      std: 0.01        # distribution with stdev 0.01 (default mean: 0)
    }
    bias_filler {
      type: "constant" # initialize the biases to zero (0)
      value: 0
    }
  }
}
```



Convolution

- Layer type: Convolution
- CPU implementation: `./src/caffe/layers/convolution_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/convolution_layer.cu`
- Parameters (ConvolutionParameter convolution_param)
 - Required
 - `num_output (c_o)`: the number of filters
 - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
 - Strongly Recommended
 - `weight_filler` [default type: 'constant' value: 0]
 - Optional
 - `bias_term` [default true]: specifies whether to learn and apply a set of additive biases to the filter outputs
 - `pad` (or `pad_h` and `pad_w`) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
 - `stride` (or `stride_h` and `stride_w`) [default 1]: specifies the intervals at which to apply the filters to the input
 - `group (g)` [default 1]: If $g > 1$, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the i th output group channels will be only connected to the i th input group channels.

这对于一个初学者来说，需要花很多时间来掌握这种规则，而且，如果只是单纯的做为一种工具，你可能不需要了解细节的组成，只需要知道总共有几层，分别是哪些层，卷积层或者 **Pooling** 层的关键参数即可。因此，本文在这个功能上通过不同的函数，将细节隐去，只显示一些宏观的，关键的内容，下图是 **Lenet** 的显示结果：



Name	Type	kernel_size	num
data	Data		
conv1	Convolution	5*5	20
pool1	Pooling	2*2	
conv2	Convolution	5*5	50
pool2	Pooling	2*2	
fc1	InnerProduct		
relu1	ReLU		
score	InnerProduct		
loss	SoftmaxWithLoss		

图 6 train.prototxt 内容显示

同时，在训练中，对于一个已有的网络，用户最常更改的内容就是 `batch_size` 的大小，考虑到这一点，我们将 `batch_size` 的值设为可以调节，用户只需根据实际情况修改。

`Solver.prototxt` 是 `caffe` 框架中用来设置深度学习的一些超参数，比如学习率，迭代次数，训练得到的结果保存路径等，相对来讲并不是太复杂，因些我们将 `solver.prototxt` 中的内容直接显示在窗口中，用户可以根据需要



修改。

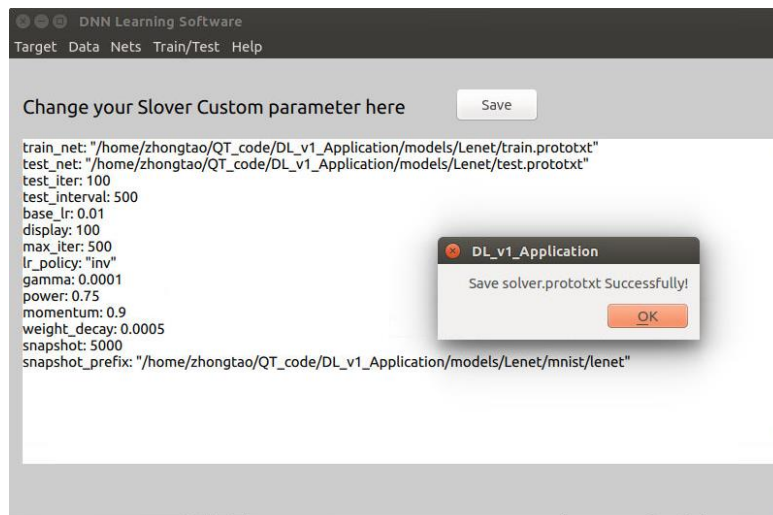


图 7 solover.prototxt 内容显示

在设置好网络结构和超参数后，就可以开始训练，可以看出，用户只需要准备好自己想训练的数据集，可以在 [caffe](#) 官网下载自己需要的网络配置文件，就可以通过本程序进行测试了。

Test.prototxt 的实现方式与 **train.prototxt** 类似，是测试过程中网络的描述。

Weightsfile 是指在测试中，除了需要提供数据集和网络结构外，还要有训练好的参数，这个功能就是将已经训练好的参数导入进来，接下来就可以测试了。

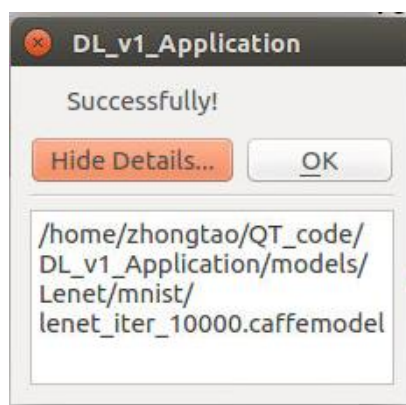


图 8 选择已训练好的参数

4.Train/test

在所有的准备工作都完成了，接下来就是训练、测试功能，实现过程实际上是将前面的所有参数提供给 **caffe** 提供的接口，就可以自动完成训练或者测试，由于编程水平有限，目前训练和测试的结果会在 **shell** 命令行中显示。此外，训练过程一般耗时非常久，少则几个小时，多则几天甚至几个月。所以本程序采用了多线程的方法，将训练或者的测试功能放在另一个线程中完成。

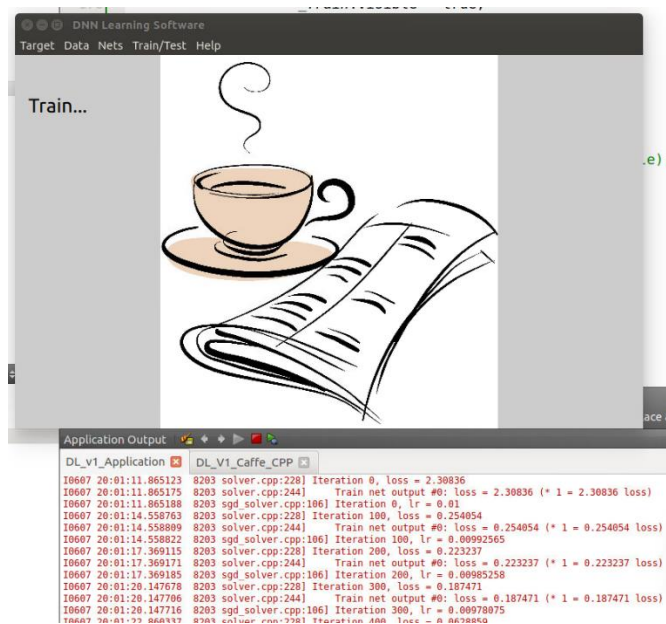
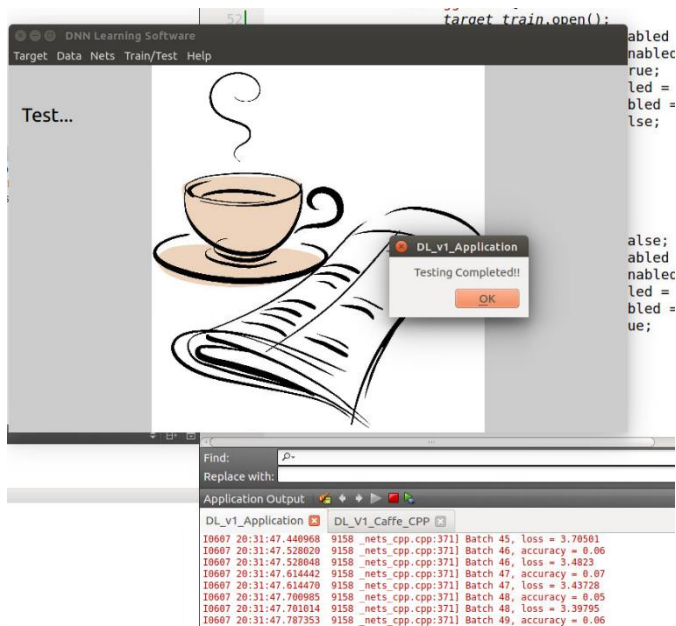


图 9

a)训练



b)测试



附录一：定制网络方法

To create a Caffe model you need to define the model architecture in a protocol buffer definition file (prototxt).

Caffe layers and their parameters are defined in the protocol buffer definitions for the project in [caffe.proto](#).

Vision Layers

- Header: `./include/caffe/vision_layers.hpp`

Vision layers usually take *images* as input and produce other *images* as output. A typical “image” in the real-world may have one color channel ($c=1$), as in a grayscale image, or three color channels ($c=3$) as in an RGB (red, green, blue) image. But in this context, the distinguishing characteristic of an image is its spatial structure: usually an image has some non-trivial height $h>1$ and width $w>1$. This 2D geometry naturally lends itself to certain decisions about how to process the input. In particular, most of the vision layers work by applying a particular operation to some region of the input to produce a corresponding region of the output. In contrast, other layers (with few exceptions) ignore the spatial structure of the input, effectively treating it as “one big vector” with dimension chw .

Convolution

- Layer type: Convolution
- CPU implementation: `./src/caffe/layers/convolution_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/convolution_layer.cu`
- Parameters (`ConvolutionParameter convolution_param`)
 - Required
 - `num_output (c_o)`: the number of filters
 - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
 - Strongly Recommended
 - `weight_filler` [default type: 'constant' value: 0]
 - Optional
 - `bias_term` [default true]: specifies whether to learn and apply a set of additive biases to the filter outputs
 - `pad` (or `pad_h` and `pad_w`) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
 - `stride` (or `stride_h` and `stride_w`) [default 1]: specifies the intervals at which to apply the filters to the input



- group (g) [default 1]: If $g > 1$, we restrict the connectivity of each filter to a subset of the input. Specifically, the input and output channels are separated into g groups, and the i th output group channels will be only connected to the i th input group channels.
- Input
 - $n * c_i * h_i * w_i$
- Output
 - $n * c_o * h_o * w_o$, where $h_o = (h_i + 2 * pad_h - kernel_h) / stride_h + 1$ and w_o likewise.
- Sample (as seen in `./models/bvlc_reference_caffenet/train_val.prototxt`)

```
• layer {
•   name: "conv1"
•   type: "Convolution"
•   bottom: "data"
•   top: "conv1"
•   # learning rate and decay multipliers for the filters
•   param { lr_mult: 1 decay_mult: 1 }
•   # learning rate and decay multipliers for the biases
•   param { lr_mult: 2 decay_mult: 0 }
•   convolution_param {
•     num_output: 96      # learn 96 filters
•     kernel_size: 11     # each filter is 11x11
•     stride: 4           # step 4 pixels between each filter application
•     weight_filler {
•       type: "gaussian" # initialize the filters from a Gaussian
•       std: 0.01        # distribution with stdev 0.01 (default mean: 0)
•     }
•     bias_filler {
•       type: "constant" # initialize the biases to zero (0)
•       value: 0
•     }
•   }
• }
```

The Convolution layer convolves the input image with a set of learnable filters, each producing one feature map in the output image.

Pooling

- Layer type: Pooling
- CPU implementation: `./src/caffe/layers/pooling_layer.cpp`



- CUDA GPU implementation: `./src/caffe/layers/pooling_layer.cu`
- Parameters (`PoolingParameter pooling_param`)
 - Required
 - `kernel_size` (or `kernel_h` and `kernel_w`): specifies height and width of each filter
 - Optional
 - `pool` [default MAX]: the pooling method. Currently MAX, AVE, or STOCHASTIC
 - `pad` (or `pad_h` and `pad_w`) [default 0]: specifies the number of pixels to (implicitly) add to each side of the input
 - `stride` (or `stride_h` and `stride_w`) [default 1]: specifies the intervals at which to apply the filters to the input
- Input
 - $n * c * h_i * w_i$
- Output
 - $n * c * h_o * w_o$, where h_o and w_o are computed in the same way as convolution.
- Sample (as seen in `./models/bvlc_reference_caffenet/train_val.prototxt`)

```
• layer {  
•   name: "pool1"  
•   type: "Pooling"  
•   bottom: "conv1"  
•   top: "pool1"  
•   pooling_param {  
•     pool: MAX  
•     kernel_size: 3 # pool over a 3x3 region  
•     stride: 2      # step two pixels (in the bottom blob) between pooling regions  
•   }  
• }
```

Local Response Normalization (LRN)

- Layer type: LRN
- CPU Implementation: `./src/caffe/layers/lrn_layer.cpp`
- CUDA GPU Implementation: `./src/caffe/layers/lrn_layer.cu`
- Parameters (`LRNParameter lrn_param`)
 - Optional
 - `local_size` [default 5]: the number of channels to sum over (for cross channel LRN) or the side length of the square region to sum over (for within channel LRN)
 - `alpha` [default 1]: the scaling parameter (see below)
 - `beta` [default 5]: the exponent (see below)



- `norm_region` [default `ACROSS_CHANNELS`]: whether to sum over adjacent channels (`ACROSS_CHANNELS`) or nearby spatial locations (`WITHIN_CHANNEL`)

The local response normalization layer performs a kind of “lateral inhibition” by normalizing over local input regions. In `ACROSS_CHANNELS` mode, the local regions extend across nearby channels, but have no spatial extent (i.e., they have shape `local_size x 1 x 1`). In `WITHIN_CHANNEL` mode, the local regions extend spatially, but are in separate channels (i.e., they have shape `1 x local_size x local_size`). Each input value is divided by $(1 + (\alpha/n) \sum_i x_i^2)^\beta$, where n is the size of each local region, and the sum is taken over the region centered at that value (zero padding is added where necessary).

im2col

`Im2col` is a helper for doing the image-to-column transformation that you most likely do not need to know about. This is used in Caffe’s original convolution to do matrix multiplication by laying out all patches into a matrix.

Loss Layers

Loss drives learning by comparing an output to a target and assigning cost to minimize. The loss itself is computed by the forward pass and the gradient w.r.t. to the loss is computed by the backward pass.

Softmax

- Layer type: `SoftmaxWithLoss`

The softmax loss layer computes the multinomial logistic loss of the softmax of its inputs. It’s conceptually identical to a softmax layer followed by a multinomial logistic loss layer, but provides a more numerically stable gradient.

Sum-of-Squares / Euclidean

- Layer type: `EuclideanLoss`

The Euclidean loss layer computes the sum of squares of differences of its two inputs, $\frac{1}{2N} \sum_{i=1}^N \|x_i - y_i\|^2$.



Hinge / Margin

- Layer type: HingeLoss
- CPU implementation: `./src/caffe/layers/hinge_loss_layer.cpp`
- CUDA GPU implementation: none yet
- Parameters (`HingeLossParameter hinge_loss_param`)
 - Optional
 - `norm` [default L1]: the norm used. Currently L1, L2
- Inputs
 - $n * c * h * w$ Predictions
 - $n * 1 * 1 * 1$ Labels
- Output
 - $1 * 1 * 1 * 1$ Computed Loss
- Samples

```
• # 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
•   }
• }
```

The hinge loss layer computes a one-vs-all hinge or squared hinge loss.



Sigmoid Cross-Entropy

`SigmoidCrossEntropyLoss`

Infogain

`InfogainLoss`

Accuracy and Top-k

`Accuracy` scores the output as the accuracy of output with respect to target – it is not actually a loss and has no backward step.

Activation / Neuron Layers

In general, activation / Neuron layers are element-wise operators, taking one bottom blob and producing one top blob of the same size. In the layers below, we will ignore the input and out sizes as they are identical:

- Input
 - $n * c * h * w$
- Output
 - $n * c * h * w$

ReLU / Rectified-Linear and Leaky-ReLU

- Layer type: ReLU
- CPU implementation: `./src/caffe/layers/relu_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/relu_layer.cu`
- Parameters (`ReLUParameter relu_param`)
 - Optional
 - `negative_slope` [default 0]: specifies whether to leak the negative part by multiplying it with the slope value rather than setting it to 0.
- Sample (as seen in `./models/bvlc_reference_caffenet/train_val.prototxt`)
 - `layer {`
 - `name: "relu1"`



- type: "ReLU"
- bottom: "conv1"
- top: "conv1"
- }

Given an input value x , The `ReLU` layer computes the output as x if $x > 0$ and $\text{negative_slope} * x$ if $x \leq 0$. When the negative slope parameter is not set, it is equivalent to the standard ReLU function of taking $\max(x, 0)$. It also supports in-place computation, meaning that the bottom and the top blob could be the same to preserve memory consumption.

Sigmoid

- Layer type: Sigmoid
- CPU implementation: `./src/caffe/layers/sigmoid_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/sigmoid_layer.cu`
- Sample (as seen in `./examples/mnist/mnist_autoencoder.prototxt`)

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

The `Sigmoid` layer computes the output as $\text{sigmoid}(x)$ for each input element x .

TanH / Hyperbolic Tangent

- Layer type: TanH
- CPU implementation: `./src/caffe/layers/tanh_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/tanh_layer.cu`
- Sample

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



The `TanH` layer computes the output as $\tanh(x)$ for each input element x .

Absolute Value

- Layer type: `AbsVal`
- CPU implementation: `./src/caffe/layers/absval_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/absval_layer.cu`
- Sample

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

The `AbsVal` layer computes the output as $\text{abs}(x)$ for each input element x .

Power

- Layer type: `Power`
- CPU implementation: `./src/caffe/layers/power_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/power_layer.cu`
- Parameters (`PowerParameter power_param`)
- Optional
 - `power` [default 1]
 - `scale` [default 1]
 - `shift` [default 0]
- Sample

```
• layer {  
•   name: "layer"  
•   bottom: "in"  
•   top: "out"  
•   type: "Power"  
•   power_param {  
•     power: 1
```



- scale: 1
- shift: 0
- }
- }

The `Power` layer computes the output as $(\text{shift} + \text{scale} * x)^{\text{power}}$ for each input element x .

BNLL

- Layer type: BNLL
- CPU implementation: `./src/caffe/layers/bnll_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/bnll_layer.cu`
- Sample

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

The `BNLL` (binomial normal log likelihood) layer computes the output as $\log(1 + \exp(x))$ for each input element x .

Data Layers

Data enters Caffe through data layers: they lie at the bottom of nets. Data can come from efficient databases (LevelDB or LMDB), directly from memory, or, when efficiency is not critical, from files on disk in HDF5 or common image formats.

Common input preprocessing (mean subtraction, scaling, random cropping, and mirroring) is available by specifying `TransformationParameters`.

Database

- Layer type: Data
- Parameters



- Required
 - `source`: the name of the directory containing the database
 - `batch_size`: the number of inputs to process at one time
- Optional
 - `rand_skip`: skip up to this number of inputs at the beginning; useful for asynchronous sgd
 - `backend` [default `LEVELDB`]: choose whether to use a `LEVELDB` or `LMDB`

In-Memory

- Layer type: `MemoryData`
- Parameters
- Required
 - `batch_size`, `channels`, `height`, `width`: specify the size of input chunks to read from memory

The memory data layer reads data directly from memory, without copying it. In order to use it, one must call `MemoryDataLayer::Reset` (from C++) or `Net.set_input_arrays` (from Python) in order to specify a source of contiguous data (as 4D row major array), which is read one batch-sized chunk at a time.

HDF5 Input

- Layer type: `HDF5Data`
- Parameters
- Required
 - `source`: the name of the file to read from
 - `batch_size`

HDF5 Output

- Layer type: `HDF5Output`
- Parameters
- Required
 - `file_name`: name of file to write to

The HDF5 output layer performs the opposite function of the other layers in this section: it writes its input blobs to disk.



Images

- Layer type: ImageData
- Parameters
 - Required
 - source: name of a text file, with each line giving an image filename and label
 - batch_size: number of images to batch together
 - Optional
 - rand_skip
 - shuffle [default false]
 - new_height, new_width: if provided, resize all images to this size

Windows

WindowData

Dummy

DummyData is for development and debugging. See DummyDataParameter.

Common Layers

Inner Product

- Layer type: InnerProduct
- CPU implementation: ./src/caffe/layers/inner_product_layer.cpp
- CUDA GPU implementation: ./src/caffe/layers/inner_product_layer.cu
- Parameters (InnerProductParameter inner_product_param)
 - Required
 - num_output (c_o): the number of filters
 - Strongly recommended
 - weight_filler [default type: 'constant' value: 0]
 - Optional
 - bias_filler [default type: 'constant' value: 0]



- `bias_term` [default `true`]: specifies whether to learn and apply a set of additive biases to the filter outputs
- Input
 - $n * c_i * h_i * w_i$
- Output
 - $n * c_o * 1 * 1$
- Sample

```
• 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"  
• }
```

The `InnerProduct` layer (also usually referred to as the fully connected layer) treats the input as a simple vector and produces an output in the form of a single vector (with the blob's height and width set to 1).

Splitting

The `split` layer is a utility layer that splits an input blob to multiple output blobs. This is used when a blob is fed into multiple output layers.



Flattening

The `Flatten` layer is a utility layer that flattens an input of shape $n * c * h * w$ to a simple vector output of shape $n * (c * h * w)$

Reshape

- Layer type: Reshape
- Implementation: `./src/caffe/layers/reshape_layer.cpp`
- Parameters (`ReshapeParameter reshape_param`)
 - Optional: (also see detailed description below)
 - `shape`
- Input
 - a single blob with arbitrary dimensions
- Output
 - the same blob, with modified dimensions, as specified by `reshape_param`
- Sample

```
•   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  
•       }  
•     }  
•   }
```

The `Reshape` layer can be used to change the dimensions of its input, without changing its data. Just like the `Flatten` layer, only the dimensions are changed; no data is copied in the process.

Output dimensions are specified by the `ReshapeParam` proto. Positive numbers are used directly, setting the corresponding dimension of the output blob. In addition, two special values are accepted for any of the target dimension values:



- **0** means “copy the respective dimension of the bottom layer”. That is, if the bottom has 2 as its 1st dimension, the top will have 2 as its 1st dimension as well, given `dim: 0` as the 1st target dimension.
- **-1** stands for “infer this from the other dimensions”. This behavior is similar to that of `-1` in *numpy*’s or `[]` for *MATLAB*’s `reshape`: this dimension is calculated to keep the overall element count the same as in the bottom layer. At most one `-1` can be used in a reshape operation.

As another example, specifying `reshape_param { shape { dim: 0 dim: -1 } }` makes the layer behave in exactly the same way as the `Flatten` layer.

Concatenation

- Layer type: `Concat`
- CPU implementation: `./src/caffe/layers/concat_layer.cpp`
- CUDA GPU implementation: `./src/caffe/layers/concat_layer.cu`
- Parameters (`ConcatParameter concat_param`)
 - Optional
 - `axis` [default 1]: 0 for concatenation along num and 1 for channels.
- Input
 - $n_i * c_i * h * w$ for each input blob i from 1 to K .
- Output
 - if `axis = 0`: $(n_1 + n_2 + \dots + n_K) * c_1 * h * w$, and all input c_i should be the same.
 - if `axis = 1`: $n_1 * (c_1 + c_2 + \dots + c_K) * h * w$, and all input n_i should be the same.
- Sample

```

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

The `Concat` layer is a utility layer that concatenates its multiple input blobs to one single output blob.



Slicing

The `slice` layer is a utility layer that slices an input layer to multiple output layers along a given dimension (currently num or channel only) with given slice indices.

- Sample

```
• 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` indicates the target axis; `slice_point` indicates indexes in the selected dimension (the number of indices must be equal to the number of top blobs minus one).