# [如何在Caffe中配置每一个层的结构](http://blog.csdn.net/lien0906/article/details/46854327)

最近刚在电脑上装好Caffe，由于神经网络中有不同的层结构，不同类型的层又有不同的参数，所有就根据Caffe官网的说明文档做了一个简单的总结。

## 1. Vision Layers

### 1.1 卷积层(Convolution)

类型：CONVOLUTION

例子

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "conv1"  type: CONVOLUTION  bottom: "data"  top: "conv1"  blobs\_lr: 1          # learning rate multiplier for the filters  blobs\_lr: 2          # learning rate multiplier for the biases  weight\_decay: 1      # weight decay multiplier for the filters  weight\_decay: 0      # weight decay multiplier for the biases  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    }  }}

**blobs\_lr:**学习率调整的参数，在上面的例子中设置权重学习率和运行中求解器给出的学习率一样，同时是偏置学习率为权重的两倍。

**weight\_decay：**

卷积层的重要参数

必须参数：

**num\_output (c\_o)：**过滤器的个数

**kernel\_size (or kernel\_h and kernel\_w)：**过滤器的大小

可选参数：

**weight\_filler [default type: 'constant' value: 0]：**参数的初始化方法

**bias\_filler：**偏置的初始化方法

**bias\_term [default true]：**指定是否是否开启偏置项

**pad (or pad\_h and pad\_w) [default 0]：**指定在输入的每一边加上多少个像素

**stride (or stride\_h and stride\_w) [default 1]：**指定过滤器的步长

**group (g) [default 1]:**If g > 1, we restrict the connectivityof each filter to a subset of the input. Specifically, the input and outputchannels are separated into g groups, and the ith output group channels will beonly connected to the ith input group channels.

通过卷积后的大小变化：

输入：n \* c\_i \* h\_i \* w\_i

输出：n \* c\_o \* h\_o \* w\_o，其中h\_o = (h\_i + 2 \* pad\_h - kernel\_h) /stride\_h + 1，w\_o通过同样的方法计算。

### 1.2 池化层（Pooling）

类型：POOLING

例子

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  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  }}

卷积层的重要参数

必需参数：

**kernel\_size (or kernel\_h and kernel\_w)：**过滤器的大小

可选参数：

**pool [default MAX]：**pooling的方法，目前有MAX, AVE, 和STOCHASTIC三种方法

**pad (or pad\_h and pad\_w) [default 0]：**指定在输入的每一遍加上多少个像素

**stride (or stride\_h and stride\_w) [default1]：**指定过滤器的步长

通过池化后的大小变化：

输入：n \* c\_i \* h\_i \* w\_i

输出：n \* c\_o \* h\_o \* w\_o，其中h\_o = (h\_i + 2 \* pad\_h - kernel\_h) /stride\_h + 1，w\_o通过同样的方法计算。

### 1.3 Local Response Normalization (LRN)

类型：LRN

Local ResponseNormalization是对一个局部的输入区域进行的归一化（激活a被加一个归一化权重（分母部分）生成了新的激活b），有两种不同的形式，一种的输入区域为相邻的channels（cross channel LRN），另一种是为同一个channel内的空间区域（within channel LRN）

计算公式：对每一个输入除以http://img.blog.csdn.net/20150120213228813?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvZGFuaWVsamlhbmZlbmc=/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center

可选参数：

**local\_size [default 5]：**对于cross channel LRN为需要求和的邻近channel的数量；对于within channel LRN为需要求和的空间区域的边长

**alpha [default 1]：**scaling参数

**beta [default 5]：**指数

**norm\_region [default ACROSS\_CHANNELS]:**选择哪种LRN的方法ACROSS\_CHANNELS 或者WITHIN\_CHANNEL

## 2. Loss Layers

深度学习是通过最小化输出和目标的Loss来驱动学习。

### 2.1 Softmax

类型: SOFTMAX\_LOSS

### 2.2 Sum-of-Squares / Euclidean

类型: EUCLIDEAN\_LOSS

### 2.3 Hinge / Margin

类型: HINGE\_LOSS

例子：

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. # L1 Normlayers {  name: "loss"  type: HINGE\_LOSS  bottom: "pred"  bottom: "label"}# L2 Normlayers {  name: "loss"  type: HINGE\_LOSS  bottom: "pred"  bottom: "label"  top: "loss"  hinge\_loss\_param {    norm: L2  }}

可选参数：

**norm [default L1]:**选择L1或者 L2范数

输入：

n \* c \* h \* wPredictions

n \* 1 \* 1 \* 1Labels

输出

1 \* 1 \* 1 \* 1Computed Loss

### 2.4 Sigmoid Cross-Entropy

类型：SIGMOID\_CROSS\_ENTROPY\_LOSS

### 2.5 Infogain

类型：INFOGAIN\_LOSS

### 2.6 Accuracy and Top-k

类型：ACCURACY   
用来计算输出和目标的正确率，事实上这不是一个loss，而且没有backward这一步。

## 3. 激励层（Activation / Neuron Layers）

一般来说，激励层是element-wise的操作，输入和输出的大小相同，一般情况下就是一个非线性函数。

### 3.1 ReLU / Rectified-Linear and Leaky-ReLU

类型: RELU

例子:

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "relu1"  type: RELU  bottom: "conv1"  top: "conv1"}

可选参数：

**negative\_slope [default 0]:**指定输入值小于零时的输出。

**ReLU是目前使用做多的激励函数，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)，而一般为当x > 0时输出x，但x <= 0时输出negative\_slope。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。

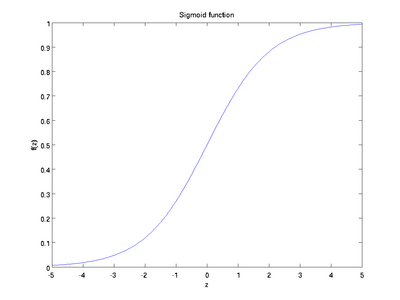
### 3.2 Sigmoid

类型: SIGMOID

例子:

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "encode1neuron"  bottom: "encode1"  top: "encode1neuron"  type: SIGMOID}

SIGMOID 层通过 sigmoid(x) 计算每一个输入x的输出，函数如下图。  


### 3.3 TanH / Hyperbolic Tangent

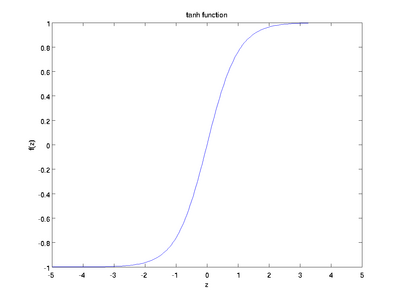
类型: TANH

例子:

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "encode1neuron"  bottom: "encode1"  top: "encode1neuron"  type: SIGMOID}

TANH层通过 tanh(x) 计算每一个输入x的输出，函数如下图。



### 3.3 Absolute Value

类型: ABSVAL

例子:

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "layer"  bottom: "in"  top: "out"  type: ABSVAL}

ABSVAL层通过 abs(x) 计算每一个输入x的输出。

### 3.4 Power

类型: POWER

例子：

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "layer"  bottom: "in"  top: "out"  type: POWER  power\_param {    power: 1    scale: 1    shift: 0  }}

可选参数：  
power [default 1]  
scale [default 1]  
shift [default 0]

POWER层通过 (shift + scale \* x) ^ power计算每一个输入x的输出。

### 3.5 BNLL

类型: BNLL

例子：

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "layer"  bottom: "in"  top: "out"  type: BNLL}

BNLL (binomial normal log likelihood) 层通过 log(1 + exp(x)) 计算每一个输入x的输出。

## 4. 数据层（Data Layers）

数据通过数据层进入Caffe，数据层在整个网络的底部。数据可以来自高效的数据库（LevelDB 或者 LMDB），直接来自内存。如果不追求高效性，可以以HDF5或者一般图像的格式从硬盘读取数据。

### 4.1 Database

类型：DATA

必须参数：

**source:**包含数据的目录名称

**batch\_size:**一次处理的输入的数量

可选参数：

**rand\_skip:**在开始的时候从输入中跳过这个数值，这在异步随机梯度下降（SGD）的时候非常有用

**backend [default LEVELDB]:** 选择使用 LEVELDB 或者 LMDB

### 4.2 In-Memory

类型: MEMORY\_DATA  
必需参数：  
batch\_size, channels, height, width: 指定从内存读取数据的大小  
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.

### 4.3 HDF5 Input

类型: HDF5\_DATA  
必要参数：  
source:需要读取的文件名

batch\_size：一次处理的输入的数量

### 4.4 HDF5 Output

类型: HDF5\_OUTPUT  
必要参数：  
file\_name: 输出的文件名

HDF5的作用和这节中的其他的层不一样，它是把输入的blobs写到硬盘

### 4.5 Images

类型: IMAGE\_DATA  
必要参数：  
source: text文件的名字，每一行给出一张图片的文件名和label  
batch\_size: 一个batch中图片的数量  
可选参数：  
rand\_skip：在开始的时候从输入中跳过这个数值，这在异步随机梯度下降（SGD）的时候非常有用  
shuffle [default false]

**new\_height, new\_width:** 把所有的图像resize到这个大小

### 4.6 Windows

类型：WINDOW\_DATA

### 4.7 Dummy

类型：DUMMY\_DATA

Dummy 层用于development 和debugging。具体参数DummyDataParameter。

## 5. 一般层（Common Layers）

### 5.1 全连接层Inner Product

类型：INNER\_PRODUCT  
例子：

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "fc8"  type: INNER\_PRODUCT  blobs\_lr: 1          # learning rate multiplier for the filters  blobs\_lr: 2          # learning rate multiplier for the biases  weight\_decay: 1      # weight decay multiplier for the filters  weight\_decay: 0      # weight decay multiplier for the biases  inner\_product\_param {    num\_output: 1000    weight\_filler {      type: "gaussian"      std: 0.01    }    bias\_filler {      type: "constant"      value: 0    }  }  bottom: "fc7"  top: "fc8"}

必要参数：

**num\_output (c\_o)：**过滤器的个数

可选参数：

**weight\_filler [default type: 'constant' value: 0]：**参数的初始化方法

**bias\_filler：**偏置的初始化方法

**bias\_term [default true]：**指定是否是否开启偏置项

通过全连接层后的大小变化：

输入：n \* c\_i \* h\_i \* w\_i

输出：n \* c\_o \* 1 \*1

### 5.2 Splitting

类型：SPLIT  
Splitting层可以把一个输入blob分离成多个输出blobs。这个用在当需要把一个blob输入到多个输出层的时候。

### 5.3 Flattening

类型：FLATTEN  
Flattening是把一个输入的大小为n \* c \* h \* w变成一个简单的向量，其大小为 n \* (c\*h\*w) \* 1 \* 1。

### 5.4 Concatenation

类型：CONCAT  
  
  
例子：

**[plain]** [view plaincopy](http://blog.csdn.net/lien0906/article/details/46854327)

1. layers {  name: "concat"  bottom: "in1"  bottom: "in2"  top: "out"  type: CONCAT  concat\_param {    concat\_dim: 1  }}

可选参数：

**concat\_dim [default 1]：**0代表链接num，1代表链接channels

通过全连接层后的大小变化：

输入：从1到K的每一个blob的大小n\_i \* c\_i \* h \* w

输出：

如果concat\_dim = 0: (n\_1 + n\_2 + ... + n\_K) \*c\_1 \* h \* w，需要保证所有输入的c\_i 相同。

如果concat\_dim = 1: n\_1 \* (c\_1 + c\_2 + ... +c\_K) \* h \* w，需要保证所有输入的n\_i 相同。

通过Concatenation层，可以把多个的blobs链接成一个blob。

### 5.5 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.

### 5.6 Elementwise Operations

类型：ELTWISE

### 5.7 Argmax

类型：ARGMAX

### 5.8 Softmax

类型：SOFTMAX

### 5.9 Mean-Variance Normalization

类型：MVN

## 6. 参考

[Caffe](http://demo.netfoucs.com/danieljianfeng/article/details/42929283)