|  |  |
| --- | --- |
| Caffe源码解析 | 文件类型：技术文档 |
| 文件编号： |
| 面向的部门： 研发中心 |
| 保密等级：高 |
| 作者： 刘凯 |
| 日期：2019-01-25 |
| 版本：1.0V |

Caffe源码解析



目 录

[1 卷积层前向计算实现 3](#_Toc536696706)

[1.1 代码结构 3](#_Toc536696707)

[1.2 im2col解析 4](#_Toc536696708)

[1.3 矩阵乘法 8](#_Toc536696709)

[2 全连接层 8](#_Toc536696710)

[3 激活层Relu 9](#_Toc536696711)

[4 池化层Pooling 10](#_Toc536696712)

[参考文献 1](#_Toc536696713)

# 卷积层前向计算实现

本节讨论caffe源码中的卷积层实现前向计算的源码。

* 1. 代码结构

Caffe的卷积层是在类ConvolutionLayer中实现的，其声明和定义在: $CAFFE\_ROOT/include/layers/conv\_layer.hpp

$CAFFE\_ROOT/src/layers/conv\_layer.cpp

ConvolutionLayer类是继承了基类BaseConvolutionLayer类，而BaseConvolutionLayer类又继承了基类Layer类。

前向网络使用ConvolutionLayer类的Forward\_cpu方法实现，而其核心代码调用BaseConvolutionLayer类的forward\_cpu\_gemm方法，其代码如下：

template <typename Dtype>

void BaseConvolutionLayer<Dtype>::forward\_cpu\_gemm

(const Dtype\* input, const Dtype\* weights,

Dtype\* output, bool skip\_im2col) {

const Dtype\* col\_buff = input;

if (!is\_1x1\_) {

if (!skip\_im2col) {

conv\_im2col\_cpu(input, col\_buffer\_.mutable\_cpu\_data());

}

col\_buff = col\_buffer\_.cpu\_data();

}

for (int g = 0; g < group\_; ++g) {

caffe\_cpu\_gemm<Dtype>(CblasNoTrans, CblasNoTrans,

conv\_out\_channels\_, group\_, conv\_out\_spatial\_dim\_,

kernel\_dim\_, (Dtype)1., weights + weight\_offset\_ \* g,

col\_buff + col\_offset\_ \* g,(Dtype)0.,

output + output\_offset\_ \* g);

}

}

从上面的源码可以看出，前向计算主要包括两个步骤：im2col和矩阵乘法。其参数含义为：

input：输入图像的指针

output：输出结果的指针

weight：权重参数的指针

skip\_im2col：是否跳过im2col

is\_1x1\_：是否为1x1的卷积

im2col将input指向的图片数据(是一个Blob)转换为一个矩阵存放在col\_buff指向的空间中。im2col用函数conv\_im2col\_cpu调用im2col.cpp文件中的im2col\_cpu函数实现(下一节解析其原理和实现代码)。矩阵乘法则将col\_buff与转化成矩阵的权重相乘，并转换为输出的格式，存放在output指向的空间中。矩阵乘调用math\_functions.hpp中的caffe\_cpu\_gemm函数实现，而caffe\_cpu\_gemm函数调用的是BLAS(Basic Linear Algebra Subprograms)库中的cblas\_sgemm函数实现。group\_是分组的参数，这里不考虑，可以默认为1。

卷积计算结束后，调用BaseConvolutionLayer类里的forward\_cpu\_bias方法加偏置。

* 1. im2col解析

模块im2col是将输入的图片按照卷积核的维度拉成一列，以方便后续和卷积核计算。具体而言，设输入图片维度为，卷积核维度为，则依次将输入图片的每个通道的图片提取的数据，并拉成一个矢量，矢量的维度为，如图1(a)所示。然后按行移动Stride个步幅并重复图1(a)的操作，如图1(b)所示。最后遍历整个图像，得到输出矩阵。输出矩阵的维度为，其中分别为卷积层输出的高和宽，其计算公式如下：





其中*P*和*S*分别为Padding和Stride。如图1(c)所示(图中维度有误)。

卷积核也相应地矢量化，得到一个的矩阵，然后和图像矩阵做矩阵乘法，得到卷积层的输出，如图1(d)所示。不过这一步不是在im2col模块中进行的。

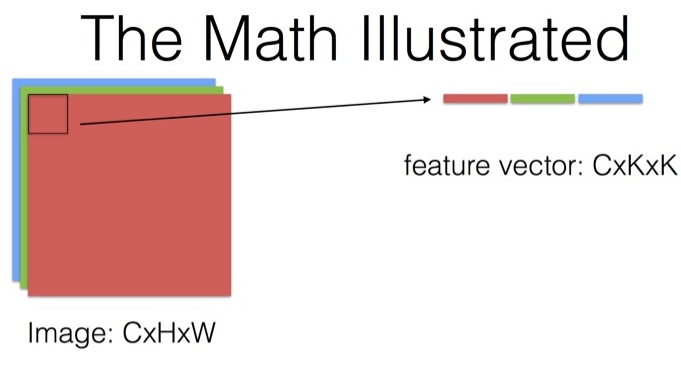


图1 (a)

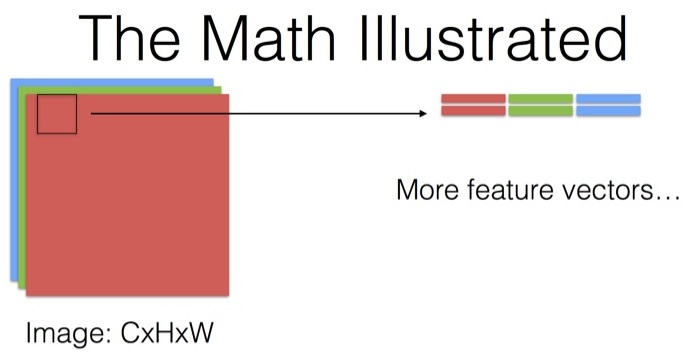


图1 (b)

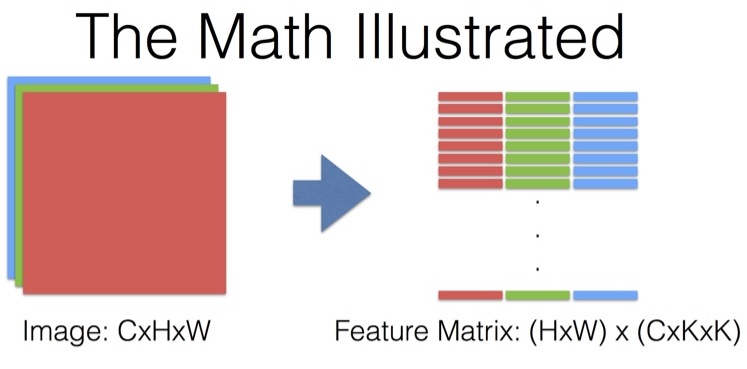


图1 (c)

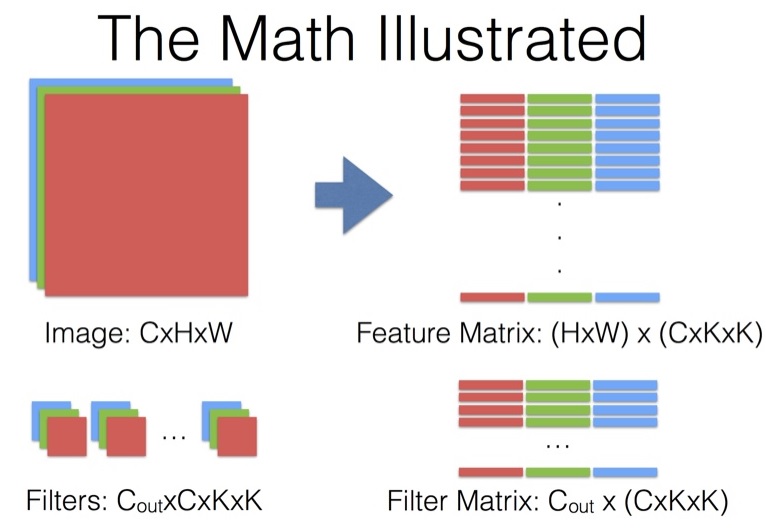


图1(d)

im2col的源码如下：

template <typename Dtype>

void im2col\_cpu(const Dtype\* data\_im, const int channels,

const int height, const int width, const int kernel\_h,

const int kernel\_w, const int pad\_h, const int pad\_w,

const int stride\_h, const int stride\_w, const int dilation\_h,

const int dilation\_w, Dtype\* data\_col) {

const int output\_h = (height + 2 \* pad\_h -

(dilation\_h \* (kernel\_h - 1) + 1)) / stride\_h + 1;

const int output\_w = (width + 2 \* pad\_w -

(dilation\_w \* (kernel\_w - 1) + 1)) / stride\_w + 1;

const int channel\_size = height \* width;

for (int channel=channels; channel--; data\_im += channel\_size) {

for (int kernel\_row=0; kernel\_row<kernel\_h; kernel\_row++) {

for (int kernel\_col=0; kernel\_col<kernel\_w; kernel\_col++) {

int input\_row = -pad\_h + kernel\_row \* dilation\_h;

for (int output\_rows=output\_h; output\_rows; output\_rows--) {

if (!is\_a\_ge\_zero\_and\_a\_lt\_b(input\_row, height)) {

for (int output\_cols=output\_w; output\_cols;

output\_cols--) {

\*(data\_col++) = 0;

}

}

else {

int input\_col = -pad\_w + kernel\_col \* dilation\_w;

for (int output\_col=output\_w; output\_col; output\_col--) {

if (is\_a\_ge\_zero\_and\_a\_lt\_b(input\_col, width)) {

\*(data\_col++) = data\_im[input\_row \* width + input\_col];

}

else {

\*(data\_col++) = 0;

}

input\_col += stride\_w;

}

}

input\_row += stride\_h;

}

}

}

}

}

这个函数的几个主要变量：

data\_im,输入图片数据的指针

channels,输入图片通道数C

height, 输入图片的高H

width, 输入图片的宽W

kernel\_h(w),卷积核的维度K

pad\_h(w), padding值P

stride\_h(w), Stride值S

dilation\_h(w), 膨胀系数D

data\_col,输出矩阵的指针

关于膨胀系数dilation是指卷积核从*K*增加到*D\*(K-1)+1*。如K=3，D=2时，卷积核就由膨胀到，中间的部分补零。图2为一个由的卷积核通过2倍膨胀到的示意图。膨胀系数一般默认为1。

im2col里有5个for循环和2个if-else判断。其中if判断里的函数is\_a\_ge\_zero\_and\_a\_lt\_b(a,b)检验是否满足，满足时返回true，否则返回false。这个是在padding不为零时，对边界的判断。从外到内第一个for表示遍历所有通道。第二个和第三个for遍历图片单通道的的区域，表示了输出单通道矩阵的某一列，同时体现了输出单通道矩阵的行数。第四个和第五个for循环表示了输出单通道矩阵的某一行，同时体现了输出单通道矩阵的列数。关于im2col的详细图解可参看[1]。

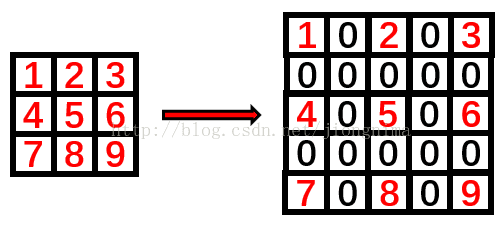


图2：卷积核2倍膨胀示意图

* 1. 矩阵乘法

矩阵乘法GEMM是general matrix matrix multiplication的意思。Caffe中的caffe\_cpu\_gemm函数原型如下：

void caffe\_cpu\_gemm<float>(const CBLAS\_TRANSPOSE TransA,

const CBLAS\_TRANSPOSE TransB, const int M, const int N,

const int K, const float alpha, const float\* A,

const float\* B, const float beta, float\* C)

其中CBLAS\_TRANSPOSE是一个枚举常量，表征了不转置、转置、共轭转置和共轭不转置四个状态。这个函数的功能是计算：C=alpha\*A\*B+beta\*C。当alpha=1且beta=0时，这个函数等价于计算矩阵乘法A\*B。矩阵A的维度为M\*K，矩阵B的维度为K\*N，矩阵C的维度为M\*N。

# 全连接层

全连接层的前向计算是在InnerProductLayer类里的Forward\_cpu方法中实现的，其源码为：

template <typename Dtype>

void InnerProductLayer<Dtype>::Forward\_cpu(

const vector<Blob<Dtype>\*>& bottom,

const vector<Blob<Dtype>\*>& top) {

const Dtype\* bottom\_data = bottom[0]->cpu\_data();

Dtype\* top\_data = top[0]->mutable\_cpu\_data();

const Dtype\* weight = this->blobs\_[0]->cpu\_data();

caffe\_cpu\_gemm<Dtype>(CblasNoTrans, transpose\_ ? CblasNoTrans :

CblasTrans, M\_, N\_, K\_, (Dtype)1.,

bottom\_data, weight, (Dtype)0., top\_data);

if (bias\_term\_) {

caffe\_cpu\_gemm<Dtype>(CblasNoTrans, CblasNoTrans, M\_, N\_,

1, (Dtype)1., bias\_multiplier\_.cpu\_data(),

this->blobs\_[1]->cpu\_data(), (Dtype)1., top\_data);

}

}

可以看到全连接层的核心计算是调用caffe\_cpu\_gemm函数，第一次计算矩阵乘，第二次是加上bias，参数bias\_term\_是判断是否需要加bias。

# 激活层Relu

激活层有很多种算法，这里只给出Relu(包括Leaky Relu)的做法。Relu是在ReLULayer类里的Forward\_cpu方法里实现的，源码为：

template <typename Dtype>

void ReLULayer<Dtype>::Forward\_cpu(

const vector<Blob<Dtype>\*>& bottom,

const vector<Blob<Dtype>\*>& top) {

const Dtype\* bottom\_data = bottom[0]->cpu\_data();

Dtype\* top\_data = top[0]->mutable\_cpu\_data();

const int count = bottom[0]->count();

Dtype negative\_slope = this->layer\_param\_.relu\_param().negative\_slope();

for (int i = 0; i < count; ++i) {

top\_data[i] = std::max(bottom\_data[i], Dtype(0))

+ negative\_slope \* std::min(bottom\_data[i], Dtype(0));

}

}

可以看到relu的核心代码是for循环中的红色部分，因为引入了参数negative\_slope，所以这段代码也可以实现leaky relu。

# 池化层Pooling

池化层在PoolingLayer类里的Forward\_cpu方法中实现。池化层只有最大池化和平均池化两种。源码如下：

template <typename Dtype>

void PoolingLayer<Dtype>::Forward\_cpu(const vector<Blob<Dtype>\*>& bottom,

const vector<Blob<Dtype>\*>& top) {

const Dtype\* bottom\_data = bottom[0]->cpu\_data();

Dtype\* top\_data = top[0]->mutable\_cpu\_data();

const int top\_count = top[0]->count();

// We'll output the mask to top[1] if it's of size >1.

const bool use\_top\_mask = top.size() > 1;

int\* mask = NULL; // suppress warnings about uninitalized variables

Dtype\* top\_mask = NULL;

// Different pooling methods. We explicitly do the switch outside the for

// loop to save time, although this results in more code.

switch (this->layer\_param\_.pooling\_param().pool()) {

case PoolingParameter\_PoolMethod\_MAX:

// Initialize

if (use\_top\_mask) {

top\_mask = top[1]->mutable\_cpu\_data();

caffe\_set(top\_count, Dtype(-1), top\_mask);

} else {

mask = max\_idx\_.mutable\_cpu\_data();

caffe\_set(top\_count, -1, mask);

}

caffe\_set(top\_count, Dtype(-FLT\_MAX), top\_data);

// The main loop

for (int n = 0; n < bottom[0]->num(); ++n) {

for (int c = 0; c < channels\_; ++c) {

for (int ph = 0; ph < pooled\_height\_; ++ph) {

for (int pw = 0; pw < pooled\_width\_; ++pw) {

int hstart = ph \* stride\_h\_ - pad\_h\_;

int wstart = pw \* stride\_w\_ - pad\_w\_;

int hend = min(hstart + kernel\_h\_, height\_);

int wend = min(wstart + kernel\_w\_, width\_);

hstart = max(hstart, 0);

wstart = max(wstart, 0);

const int pool\_index = ph \* pooled\_width\_ + pw;

for (int h = hstart; h < hend; ++h) {

for (int w = wstart; w < wend; ++w) {

const int index = h \* width\_ + w;

if (bottom\_data[index] > top\_data[pool\_index]) {

top\_data[pool\_index] = bottom\_data[index];

if (use\_top\_mask) {

top\_mask[pool\_index] = static\_cast<Dtype>(index);

} else {

mask[pool\_index] = index;

}

}

}

}

}

}

// compute offset

bottom\_data += bottom[0]->offset(0, 1);

top\_data += top[0]->offset(0, 1);

if (use\_top\_mask) {

top\_mask += top[0]->offset(0, 1);

} else {

mask += top[0]->offset(0, 1);

}

}

}

break;

case PoolingParameter\_PoolMethod\_AVE:

for (int i = 0; i < top\_count; ++i) {

top\_data[i] = 0;

}

// The main loop

for (int n = 0; n < bottom[0]->num(); ++n) {

for (int c = 0; c < channels\_; ++c) {

for (int ph = 0; ph < pooled\_height\_; ++ph) {

for (int pw = 0; pw < pooled\_width\_; ++pw) {

int hstart = ph \* stride\_h\_ - pad\_h\_;

int wstart = pw \* stride\_w\_ - pad\_w\_;

int hend = min(hstart + kernel\_h\_, height\_ + pad\_h\_);

int wend = min(wstart + kernel\_w\_, width\_ + pad\_w\_);

int pool\_size = (hend - hstart) \* (wend - wstart);

hstart = max(hstart, 0);

wstart = max(wstart, 0);

hend = min(hend, height\_);

wend = min(wend, width\_);

for (int h = hstart; h < hend; ++h) {

for (int w = wstart; w < wend; ++w) {

top\_data[ph \* pooled\_width\_ + pw] +=

bottom\_data[h \* width\_ + w];

}

}

top\_data[ph \* pooled\_width\_ + pw] /= pool\_size;

}

}

// compute offset

bottom\_data += bottom[0]->offset(0, 1);

top\_data += top[0]->offset(0, 1);

}

}

break;

case PoolingParameter\_PoolMethod\_STOCHASTIC:

NOT\_IMPLEMENTED;

break;

default:

LOG(FATAL) << "Unknown pooling method.";

}

}

池化的代码虽然长，但结构很简单，就是用switch-case实现了最大池化和平均池化的操作。

对于最大池化，先将结果初始化为负无穷，从外到内的四个for循环是按照[N, C, H, W]的顺序读取池化核范围的数据，计算开始和结束的坐标。最内的两层for循环则是依次判断并保存最大值及其坐标。

对于平均池化，for循环的操作类似，只是输出的初始化为0，且最内的两层for循环是计算池化核内所有数的平均值。

参考文献

1. <https://blog.csdn.net/jiongnima/article/details/69736844>