Layer层的中的卷积类

卷积类是caffe中最核心的一个类，但是相比于全连接层、损失函数的前向传播和反向传播过程，卷积类的前向传播和反向传播可能一开始并没有那么好理解，因为里面有涉及了一个非常核心的函数im2col，这个函数可以说是caffe框架中核心的核心。关于im2col函数我在《im2col函数》一节中已经详细的分析了，在看卷积类的代码之前必须要搞懂im2col函数。那么在本节中的分析中就假设大家已经了解了im2col函数的具体功能了。

卷积类的正向传播和反向传播推导

卷积参数的计算：

下面是卷积的计算公式，这个推导考虑的比较全面，充分的考虑了一般的情况。

输入的feature map的size为：$N \times C \times H \times W$。

卷积核kernel的size为：$N_{kernel} \times C \times kernel\_H \times kernel\_W$。

pad的size为：$pad\_H \times pad\_W$。

stride的size为：$stride\_H \times stride\_W$。

dilation的size为：$dilation\_H \times dilation\_W$。

获得的feature map的size为：

$$N\_output = N \tag{1}$$

$$C\_output = N_{kernel} \tag{2}$$

$$H\_output = (H + 2 * pad\_H - (dilation\_H * (kernel\_H - 1) + 1))/stride\_H + 1 \tag{3}$$

$$W\_output = (W + 2 * pad\_W - (dilation\_W * (kernel\_H - 1) + 1))/stride\_W + 1 \tag{4}$$

根据im2col函数的实现，我们知道在做卷积计算的时候其实是将输入的feature map拉成一个大矩阵，当然我们的所有的卷积核的参数也会拉成一个大矩阵。然后这两个大矩阵做矩阵乘法获得我们输出的结果。那下面我就利用矩阵来解释卷积的前向传播过程和反向传播过程。

首先我们卷积层输入的数据为：$A^{[l-1]} \in (N, C, H, W)$。那我们将$A^{[l-1]}$通过im2col函数变换为：$A^{[l-1]}_{im2col} \in (N, C \times kernel\_H \times kernel\_W, H\_output \times W\_output)$。

注意$A^{[l-1]}_{im2col}$现在已经是一个三维张量了。那我们的权重参数$W^{[l]} \in (N_{kernel}, C \times kernel_H \times kernel_W)$。这个$W^{[l]}$已经是一个矩阵了。我们的偏置项$b^{[l]} \in (1, N_{kernel})$。它是一个向量。

由于$A^{[l-1]}_{im2col}$是一个三维张量，而$W^{[l]}$是一个二维矩阵，所以我们并不能简单的做矩阵乘法就可以得到结果，而是做成一个for循环进行矩阵乘法，那我们具体看一下卷积的前向传播和反向传播算法的伪代码吧。

前向传播算法：

for i = 0 to N - 1 **do**

做im2col函数操作：$A^{[l-1]}_{im2col,i} = im2col(A^{[l-1]}_i)$。

做矩阵乘法运算：$Z_i^{[l]} = W^{[l]} A_{im2col,i}^{[l-1]} + b^{[l]}$

注意这里将$b^{[l]}$加到整个feature map中去需要做一个broadcast的动作。

end

对于每个$Z_i^{[l]}$来说其维度为$(N_{kernel}, H_output \times W_output)$，那对于$Z^{[l]}$来说其维度为$(N, N_{kernel}, H_output \times W_output)$

这么一看感觉卷积的前向传播过程和全连接神经网络的前向传播过程非常相似，只是多了一个im2col的转换过程。那么后面我们来看一下其反向传播的过程。

反向传播算法：

for i = 0 to N - 1 **do**

对权重$W^{[l]}$求梯度：$dW^{[l]} = \frac{\partial E}{\partial W^{[l]}} += dZ_i^{[l]} A_{im2col,i}^{[l-1]T}$

对偏置$b^{[l]}$求梯度：

$$db^{[l]} = \frac{\partial E}{\partial b^{[l]}} += \sum_{j=1}^{H\_output \times W\_output} dZ_i^{[l](j)} \tag{5}$$

注意求$db^{[l]}$的梯度的时候有一个规约的操作。

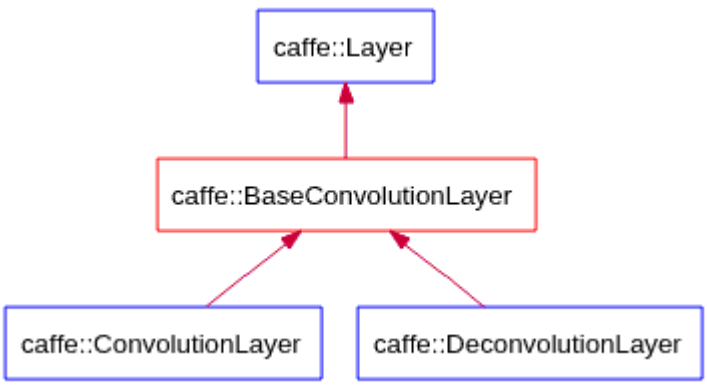
对$A_{im2col,i}^{[l-1]}$求梯度：$dA_{im2col,i}^{[l-1]} = W^{[l]T} dZ_i^{[l]}$

那对$A^{[l-1]}$求梯度：$dA^{[l-1]} = \frac{\partial E}{\partial A^{[l-1]}} += col2im(dA_{im2col,i}^{[l-1]})$

end

卷积函数的反向传播就稍微比比较复杂一点，对于$dW^{[l]}$和$db^{[l]}$以及$dA^{[l-1]}$都是累加的，因为一个batch中的每个feature map都是会对梯度产生影响的，所以这里是累加的方式。

代码分析



首先我们来看一下卷积类的继承体系，一个基础卷积类BaseConvolutionLayer派生了两个子类ConvolutionLayer和DeconvolutionLayer，即卷积核反卷积操作，关于反卷积操作后面会介绍到。在这三个类中最重要的是BaseConvolutionLayer这个类，具体的前向和反向传播算法都是在此类中实现，因为卷积类的代码量非常大，我们在这里只是具体的分析了基本卷积类的前向传播过程和反向传播过程，关于1×1的卷积和反卷积的代码不再做分析，如果搞懂了基本卷积类的代码，其他的代码理解起来也是比较容易的。

Forward_cpu函数分析

```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  //注意我们的权重是在blobs_[0]中进行存储的
  const Dtype* weight = this->blobs_[0]->cpu_data();
  for (int i = 0; i < bottom.size(); ++i) {//一般来说只有一个blob输入进来
    const Dtype* bottom_data = bottom[i]->cpu_data();
    Dtype* top_data = top[i]->mutable_cpu_data();
    for (int n = 0; n < this->num_; ++n) {
  //在这里卷积的过程并不能通过一个大矩阵相乘计算来得到，这里将batch中的每个feature map进行前向
  //的卷积计算，所以这里有一个for循环。
      this->forward_cpu_gemm(bottom_data + n * this->bottom_dim_, weight,
          top_data + n * this->top_dim_);
      if (this->bias_term_) {//如果有偏置项应该添加偏置项。
        const Dtype* bias = this->blobs_[1]->cpu_data();
        this->forward_cpu_bias(top_data + n * this->top_dim_, bias);
      }
    }
  }
}
```

那这里调用了两个封装函数即forward_cpu_gemm和forward_cpu_bias函数，这两个函数是在BaseConvolutionLayer这个类中定义的，我们来看一下吧。

Forward_cpu_gemm函数

首先我们先将输入的feature maps调用im2col函数展开成一个大矩阵，那该矩阵的维度为$(C \times kernel_H \times kernel_W, H_output \times W_output)$。当然conv_im2col_cpu函数也是一个分装函数，这就是调用im2col函数，这里就不去看了。

```
  const Dtype* col_buff = input;
  if (!is_1x1_) {
    if (!skip_im2col) {   //关于1×1卷积后面在分析
      conv_im2col_cpu(input, col_buffer_.mutable_cpu_data());
    }
    col_buff = col_buffer_.cpu_data();
  }
```

将权重矩阵和前面展开的im2col矩阵做矩阵乘法运算，得到结果的维度为$(N_{kernel}, H_output \times W_output)$。

那这里就做好了一个feature maps的前向传播卷积过程。

```
  for (int g = 0; g < group_; ++g) { //group_的默认值为1
    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);
  }
```

Forward_cpu_bias函数

做好卷积之后，如果需要添加偏置项，我们给其添加偏置项，注意在这里使用了一个bias_multiplier_做了一个broadcast的操作，既让feature map上面都加上相对应的bias。

```
275 template <typename Dtype>
276 void BaseConvolutionLayer<Dtype>::forward_cpu_bias(Dtype* output,
277     const Dtype* bias) {
278   caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasNoTrans, num_output_,
279       out_spatial_dim_, 1, (Dtype)1., bias, bias_multiplier_.cpu_data(),
280       (Dtype)1., output);
```

Backward_cpu函数

反向传播主要有三部分参数需要反向传播，第一部分偏置bias的参数，第二部分是权重weight的参数，第三部分是bottom data，它们分别调用BaseConvlutionLayer中的backward_cpu_bias函数，weight_cpu_gemm函数和backward_cpu_gemm函数，后面我们具体看一下这几个函数是怎么样的。

```
42 template <typename Dtype>
43 void ConvolutionLayer<Dtype>::Backward_cpu(const vector<Blob<Dtype>*>& top,
44       const vector<bool>& propagate_down, const vector<Blob<Dtype>*>& bottom) {
45   const Dtype* weight = this->blobs_[0]->cpu_data();
46   Dtype* weight_diff = this->blobs_[0]->mutable_cpu_diff();
47   for (int i = 0; i < top.size(); ++i) {
48     const Dtype* top_diff = top[i]->cpu_diff();
49     const Dtype* bottom_data = bottom[i]->cpu_data();
50     Dtype* bottom_diff = bottom[i]->mutable_cpu_diff();
51     // Bias gradient, if necessary.
52     if (this->bias_term_ && this->param_propagate_down_[1]) {
53       Dtype* bias_diff = this->blobs_[1]->mutable_cpu_diff();
54       for (int n = 0; n < this->num_; ++n) {
    //对偏置项进行反向传播
55         this->backward_cpu_bias(bias_diff, top_diff + n * this->top_dim_);
56       }
57     }
58     if (this->param_propagate_down_[0] || propagate_down[i]) {
59       for (int n = 0; n < this->num_; ++n) {
60         // gradient w.r.t. weight. Note that we will accumulate diffs.
    //对权重进行反向传播
61         if (this->param_propagate_down_[0]) {
62           this->weight_cpu_gemm(bottom_data + n * this->bottom_dim_,
63               top_diff + n * this->top_dim_, weight_diff);
64         }
65         // gradient w.r.t. bottom data, if necessary.
66         if (propagate_down[i]) {
    //对bottom data进行反向传播
67           this->backward_cpu_gemm(top_diff + n * this->top_dim_, weight,
68               bottom_diff + n * this->bottom_dim_);
69         }
70       }
71     }
72   }
73 }
```

backward_cpu_bias函数

需要注意的一点是因为我们的bias是共享参数，$dZ_i^{[l]}$对$db^{[l]}$的梯度都有贡献，所以反向传播的时候是采用累加的方式计算梯度。但是对于单个$dZ_i^{[l]}$反向传播需要进行归约求和操作。

```
317 template <typename Dtype>
318 void BaseConvolutionLayer<Dtype>::backward_cpu_bias(Dtype* bias,
319     const Dtype* input) {
320   caffe_cpu_gemv<Dtype>(CblasNoTrans, num_output_, out_spatial_dim_, 1.,
321       input, bias_multiplier_.cpu_data(), 1., bias);
322 }
```

weight_cpu_gemm函数

那下面的代码就是来实现这个操作

$$dW^{[l]} = \frac{\partial E}{\partial W[l]} += dZ_i^{[l]} A_{im2col,i}^{[l-1]T} \tag{6}$$

```
301 template <typename Dtype>
302 void BaseConvolutionLayer<Dtype>::weight_cpu_gemm(const Dtype* input,
303     const Dtype* output, Dtype* weights) {
304   const Dtype* col_buff = input;
305   if (!is_1x1_) {   //进行im2col操作
306     conv_im2col_cpu(input, col_buffer_.mutable_cpu_data());
307     col_buff = col_buffer_.cpu_data();
308   }
```

注意我们后面的矩阵是进行了转置操作。

```
309   for (int g = 0; g < group_; ++g) {
310     caffe_cpu_gemm<Dtype>(CblasNoTrans, CblasTrans, conv_out_channels_ / group_,
311         kernel_dim_, conv_out_spatial_dim_,
312         (Dtype)1., output + output_offset_ * g, col_buff + col_offset_ * g,
313         (Dtype)1., weights + weight_offset_ * g);
314   }
315 }
```

backward_cpu_gemm函数

这里分两步操作，第一步求$A_{im2col,i}^{[l-1]}$求梯度即：$dA_{im2col,i}^{[l-1]} = W^{[l]T} dZ_i^{[l]}$

```
290   for (int g = 0; g < group_; ++g) {
291     caffe_cpu_gemm<Dtype>(CblasTrans, CblasNoTrans, kernel_dim_,
292         conv_out_spatial_dim_, conv_out_channels_ / group_,
293         (Dtype)1., weights + weight_offset_ * g, output + output_offset_ * g,
294         (Dtype)0., col_buff + col_offset_ * g);
295   }
```

第二步求$A^{[l-1]}$求梯度：$dA^{[l-1]} = \frac{\partial E}{\partial A^{[l-1]}} += col2im(dA^{[l-1]}_{im2col,i})$，因此这里需要调用col2im函数。col2im函数很简单，就是im2col函数的反过来调用。但是注意的是col2im函数是将col大矩阵的数据累加回到im上。

```
296   if (!is_1x1_) {
297     conv_col2im_cpu(col_buff, input);
298   }
```

1×1的卷积

如果卷积核的空间大小为1×1，那它的前向传播和反向传播会是怎么样的呢？它会比普通卷积更加节省计算量，而且不会有im2col函数的调用，因为1×1卷积过程可以直接用矩阵乘法来表示，我们简单看一下维度信息。

输入的数据$A^{[l-1]} \in (N, C, H, W)$，我们看一下卷积核的维度信息$(N_{kernels}，C，1，1)$，也可以是$(N_{kernels}，C)$。所以$(N_{kernels}，C) \times (C, H \times W) = (N_{kernels}, H \times W)$。哈哈，这就是天然的一个矩阵计算过程。由于这个1×1卷积的前向传播和反向传播过程非常简单，我们就不细看了。

反卷积

一般的我们的卷积类是会将一个feature map的尺寸给缩小，而反卷积类则会将一个feature map的尺寸给扩大。其实在前面卷积的反向传播中就是做了这么扩大feature map尺寸的动作。那么我们可以通过卷积的反向函数作为反卷积的前向传播函数，但是维度信息得自己清楚。其实如果我们理解了卷积的过程就是一个矩阵乘法的过程，那么反卷积的推导也是非常容易的（但在这里我这边就先不推导了，如果后有用到反卷积在去认真推导一遍公式）。