Convolutional Layer and Its Matrix Implementation

YINGHAI LU

There are quite some tutorials [1, 2, 3] and textbooks [4] explaining the mathematics of convolutional neural networks. However, in order to implement the CNN efficiently, we usually cast it into matrix operations. Few articles discuss this transformation process. [1] does a good job at the forward propagation pass but the backward propagation part is missing. In addition, many of them discuss simplified model with zero padding and 1-stride. When I was reading the code of *CAFFE2*, I found myself constantly needing to figure out the all the parameters and all the indexing into the tensors. This notes tries to bridge the gap and provides a reference for matrix implementation of convolutional layer, consider various parameters and flavors.

1 CONFIGURATION

The input to the convolutional layer is a 4D tensor $\mathbf{X} \in \mathbb{R}^{N \times C \times H_X \times W_X}$ (assuming NCHM format), which means a batch of N images of size $H \times W$, where each image has C channels. A set of M filters of size $C \times H_K \times W_K$ will be applied to those input and generate the output Y. The filters are thus represented by a tensor $\mathbf{W} \in \mathbb{R}^{M \times C \times H_K \times W_K}$. Optionally, we can have a bias vector $\mathbf{b} \in \mathbb{R}^M$, which offsets the result of the convolution. The output tensor $\mathbf{Y}$ is of size $N \times M \times H_Y \times W_Y$, where the output image (activation map) dimension $H_Y \times W_Y$ is determined by configurations such as padding and stride and etc.

For now, we assume zero-padding and stride of size 1 for simplicity. We don't take group filters into account either. Later, we will see how to extend it. We also assume $N = 1$ and omit the n subscript.

2 FORWARD PROPAGATION

For a certain filter m, we drag it through the image to apply the convolution. Each time, we apply it onto a $H_K \times W_K$ tile of the image with C channels and produce one element (pixel) in the output map:

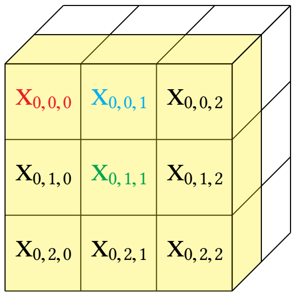
$$\mathbf{Y}_{m,i,j} = \sum_{c=0}^{C-1} \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \mathbf{W}_{m,c,p,q} \mathbf{X}_{c,i+p,j+q} + \mathbf{b}_m \tag{1}$$

The above equation can be naturally converted to maxtrix multiplication. Let's first ignore the bias and focus on the convolution term. Given an input sample $\mathbf{X} \in \mathbb{R}^{C \times H_X \times W_X}$, we are going to apply one filter $H_Y \times W_Y$ times by going from left to right and then from top to bottom. We introduce an operator $Im2Col()$ which takes $\mathbf{Y}$ as input and creates a 2D matrix $\mathbf{B} \in \mathbb{R}^{(C \times H_K \times W_K) \times (H_Y \times W_Y)}$, where each column $B_{n,:,j}$ is made by reshaping the input area where the filter is going to be applied to into a $C \times H_K \times W_K$ column. As we are moving the filter, we are going to generate such a column ($H_Y \times W_Y$ times from left to right, top to bottom of the image. Here is an example of how we generate $\mathbf{B}$ from a $2 \times 3 \times 3$ input $\mathbf{X}$ with a $2 \times 2 \times 2$ filter kernel.

With $\mathbf{B}$, if we think the weight tensor as a 2D matrix of $M \times (C \times H_K \times W_K)$, where each row is just the filter map m flattened, we can easily get the output of applying $\mathbf{W}$ to $\mathbf{X}$ as

$$\mathbf{Y} = \mathbf{WB} \tag{2}$$

where $\mathbf{Y} \in \mathbb{R}^{M \times (H_Y \times W_Y)}$.



(a) Input $\mathbf{X}$ of $2 \times 3 \times 3$

$$\begin{pmatrix} X_{0,0,0} & X_{0,0,1} & X_{0,1,0} & X_{0,1,1} \\ X_{0,0,1} & X_{0,0,2} & X_{0,1,1} & X_{0,1,2} \\ X_{0,1,0} & X_{0,1,1} & X_{0,2,0} & X_{0,2,1} \\ X_{0,1,1} & X_{0,1,2} & X_{0,2,1} & X_{0,2,2} \\ X_{1,0,0} & X_{1,0,1} & X_{1,1,0} & X_{1,1,1} \\ X_{1,0,1} & X_{1,0,2} & X_{1,1,1} & X_{1,1,2} \\ X_{1,1,0} & X_{1,1,1} & X_{1,2,0} & X_{1,2,1} \\ X_{1,1,1} & X_{1,1,2} & X_{1,2,1} & X_{1,2,2} \end{pmatrix}$$

(b) $\mathbf{B} = Im2Col(\mathbf{X})$ of 8×4

Fig. 1. An example of $Im2Col()$

Now we amend $\mathbf{Y}$ to add bias term. What we want to do is to add $\mathbf{b}_i$ to each element in row i of $\mathbf{Y}$. This can be done by generating a increament matrix and do element-wise add to $\mathbf{Y}$:

$$\mathbf{Y}' = \mathbf{Y} + \mathbf{b}\mathbf{u}^T \tag{3}$$

where $\mathbf{u} \in \mathbb{R}^{(H_Y \times W_Y) \times 1}$ is a vector filled with 1s. By doing an outer-product between $\mathbf{b}$ and $\mathbf{u}$, we strectch the incremental matrix we want and add it to $\mathbf{Y}$. Note that most numerical libraries offser GEMM routine which does operation in above equation in one call.

Applying this to each of the input batch, we can get $\mathbf{Y}$ for each $n \in \{0, \ldots, N-1\}$.

3 BACKWARD PROPAGATION

For backward propagation, we are interested in generating the gradients for inputs ($d\mathbf{X}$), for weights ($d\mathbf{W}$) and for the bias terms ($d\mathbf{b}$). In addition to $\mathbf{X}$ and $\mathbf{W}$, we also know the output gradient back-propagated from next layer $d\mathbf{Y} \in \mathbb{R}^{N \times M \times H_Y \times W_Y}$. We will focus on one sample of the input batch N and hence omit the index n in the following discussion.

The gradient of weight $\mathbf{W}_{m,c,p,q}$ with respect to the training error is about how much a perturbation in $\mathbf{W}_{m,c,p,q}$ is going to affect the output. It's worth noticing how much $\mathbf{W}_{m,c,p,q}$ is involved in the computation. Basically, for each filter m, if we have zero padding and stride of 1, $\mathbf{W}_{m,c,p,q}$ will be involved in the computation of each pixel of the output activation map with respect to a specific filter and there are $H_Y \times W_Y$ of them. With that, applying the chain rule on the gradient and we get:

$$\frac{\partial E}{\partial \mathbf{W}_{m,c,p,q}} = \sum_{i=0}^{H_Y-1} \sum_{j=0}^{W_Y-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \frac{\partial \mathbf{Y}_{m,i,j}}{\partial \mathbf{W}_{m,c,p,q}} \tag{4}$$

Substituting $\mathbf{Y}_{m,i,j}$ with (1), we have

$$\frac{\partial E}{\partial \mathbf{W}_{m,c,p,q}} = \sum_{i=0}^{H_Y-1} \sum_{j=0}^{W_Y-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \mathbf{X}_{c,i+p,j+q} \tag{5}$$

For a given input of $d\mathbf{Y}$ from a batch of N, we can treat it as a matrix $d\mathbf{Y}$ of size $M \times (H_Y \times W_Y)$. For each row m, each column represents an element $\mathbf{Y}_{m,i,j}$, flatterned into one row. Remember the buffer matrix $\mathbf{B} = Im2Col(\mathbf{X})$ is a matrix of size $(C \times H_K \times W_K) \times (H_Y \times W_Y)$, we have

$$d\mathbf{W} = d\mathbf{Y}\mathbf{B}^T \tag{6}$$

To see why, you can expand the matrix multiplication and verify that each element operation corresponds exactly to (5).

Next, let's check the gradient of bias $\mathbf{b}_m$. Similar to $\mathbf{W}_{m,c,p,q}$, $\mathbf{b}_m$ contributes to the computation of each of the pixels in an activation map of size $H_Y \times W_Y$. Similarly, applying the chain rule and we have

$$\frac{\partial E}{\partial \mathbf{b}_m} = \sum_{i=0}^{H_Y-1} \sum_{j=0}^{W_Y-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \frac{\partial \mathbf{Y}_{m,i,j}}{\partial \mathbf{b}_m} \tag{7}$$

From (1), we know that $\frac{\partial \mathbf{Y}_{m,i,j}}{\partial \mathbf{b}_m} = 1$. Therefore we have

$$d\mathbf{b} = d\mathbf{Y}\mathbf{u} \tag{8}$$

where $\mathbf{u}$ is the vector with all ones, as we have already seen in (3).

Finally, we still need to compute the gradient of input, $d\mathbf{X}$. Using chain rule, the gradient of each input $\mathbf{X}_{c,i,j}$ is

$$\frac{\partial E}{\partial \mathbf{X}_{c,i,j}} = \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \sum_{m=0}^{M-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i-p,j-q}} \frac{\partial \mathbf{Y}_{m,i-p,j-q}}{\partial \mathbf{X}_{c,i,j}} \tag{9}$$

Substituting (1) into the above euqation and we have

$$\frac{\partial E}{\partial \mathbf{X}_{c,i,j}} = \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \sum_{m=0}^{M-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i-p,j-q}} \mathbf{W}_{m,c,p,q} \tag{10}$$

Note that the ranges of the three levels of summation show intuitively how a pixel $\mathbf{X}_{c,i,j}$ contributes to the computation of output activation map $\mathbf{Y}$. The outer two levels show that each pixel will be involved in computing (at most) $H_K \times W_K$ pixels in the activation map of one filtering, as is shown by the highlighted elements in Figure (1). And the innermost summation means that there are M such filters. First, we can tackle the the innermost summation. Considering the following matrix $\mathbf{B}^* \in \mathbb{R}^{C \times H_K \times W_K} \times (H_Y \times W_Y)$

$$\mathbf{B}^* = \mathbf{W}^T d\mathbf{Y} \tag{11}$$

where each element in B^* corresponds to a combination of $\sum_{m=0}^{M-1} \mathbf{W}_{m,c,p,q}\mathbf{Y}_{m,i,j}$. Basically, B^* gives the all the information we need to compute $d\mathbf{X}_{c,i,j}$ and we can rewrite (10) to

$$\frac{\partial E}{\partial \mathbf{X}_{c,i,j}} = \sum_{(a,b)\in A} \mathbf{B}^*_{a,b} \tag{12}$$

The problem is how to pick and choose the set A so that the elements we choose satisfies (10). Notice the structural similarity between $\mathbf{B}^*$ and $\mathbf{B}$ in Figure 1. Both of them are of the same size and the elements are arranged in the same way. Since $\mathbf{B} = Im2Col(X)$, we can actually obtain $d\mathbf{X}$ from $\mathbf{B}^*$ through a dual process of $Im2Col()$, which we call $Col2Im()$. What $Col2Im()$ does is the reverse process of $Im2Col()$, taking each column of the input matrix and patch it back to the output matrix of the same size of the input. Since patches overlap with each other, the values are added up, which corresponds to the summation in (12). The following Figure shows an example of how we reconstruct an element in $d\mathbf{X}$ from $\mathbf{B}^*$, where the input and kernl size are the same as the example in Figure (1). The colored boxes show how each column in $\mathbf{B}^*$ is stretched into a tile of pixels in $d\mathbf{X}$. Note how the process is exactly the reverse of that in Figure 1.

$$\begin{pmatrix}
\mathbf{W}_{0,0,0}d\mathbf{Y}_{0,0} & \mathbf{W}_{0,0,0}d\mathbf{Y}_{0,1} & \mathbf{W}_{0,0,0}d\mathbf{Y}_{1,0} & \mathbf{W}_{0,0,0}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{0,0,1}d\mathbf{Y}_{0,0} & \mathbf{W}_{0,0,1}d\mathbf{Y}_{0,1} & \mathbf{W}_{0,0,1}d\mathbf{Y}_{1,0} & \mathbf{W}_{0,0,1}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{0,1,0}d\mathbf{Y}_{0,0} & \mathbf{W}_{0,1,0}d\mathbf{Y}_{0,1} & \mathbf{W}_{0,1,0}d\mathbf{Y}_{1,0} & \mathbf{W}_{0,1,0}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{0,1,1}d\mathbf{Y}_{0,0} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{0,1} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{1,0} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{1,0,0}d\mathbf{Y}_{0,0} & \mathbf{W}_{1,0,0}d\mathbf{Y}_{0,1} & \mathbf{W}_{1,0,0}d\mathbf{Y}_{1,0} & \mathbf{W}_{1,0,0}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{1,0,1}d\mathbf{Y}_{0,0} & \mathbf{W}_{1,0,1}d\mathbf{Y}_{0,1} & \mathbf{W}_{1,0,1}d\mathbf{Y}_{1,0} & \mathbf{W}_{1,0,1}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{1,1,0}d\mathbf{Y}_{0,0} & \mathbf{W}_{1,1,0}d\mathbf{Y}_{0,1} & \mathbf{W}_{1,1,0}d\mathbf{Y}_{1,0} & \mathbf{W}_{1,1,0}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{1,1,1}d\mathbf{Y}_{0,0} & \mathbf{W}_{1,1,1}d\mathbf{Y}_{0,1} & \mathbf{W}_{1,1,1}d\mathbf{Y}_{1,0} & \mathbf{W}_{1,1,1}d\mathbf{Y}_{1,1}
\end{pmatrix}$$

(a) $\mathbf{B}^* = \mathbf{W}^T d\mathbf{Y}$ of 8×4

$$\begin{pmatrix}
\mathbf{W}_{0,0,0}d\mathbf{Y}_{0,0} & \mathbf{W}_{0,0,1}d\mathbf{Y}_{0,0} + \mathbf{W}_{0,0,0}d\mathbf{Y}_{0,1} & \mathbf{W}_{0,0,1}d\mathbf{Y}_{0,1} \\
\mathbf{W}_{0,1,0}d\mathbf{Y}_{0,0} + \mathbf{W}_{0,0,0}d\mathbf{Y}_{1,0} & \begin{matrix}\mathbf{W}_{0,1,1}d\mathbf{Y}_{0,0} + \mathbf{W}_{0,1,0}d\mathbf{Y}_{0,1} \\ +\mathbf{W}_{0,0,1}d\mathbf{Y}_{1,0}+\mathbf{W}_{0,0,0}d\mathbf{Y}_{1,1}\end{matrix} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{0,1} + \mathbf{W}_{0,0,1}d\mathbf{Y}_{1,1} \\
\mathbf{W}_{0,1,0}d\mathbf{Y}_{1,0} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{1,0} + \mathbf{W}_{0,1,0}d\mathbf{Y}_{1,1} & \mathbf{W}_{0,1,1}d\mathbf{Y}_{1,1}
\end{pmatrix}$$

(b) First channel of $d\mathbf{X} = Col2Im(\mathbf{B}^*)$ of $2 \times 3 \times 3$

Fig. 2. An example of $Col2Im()$

4 DIFFERENT FLAVORS

Normally, convulational layer will be configured to be more than just zero padding and stride 1. Hence, we discuss how to take those parameters into consideration.

4.1 Padding, Stride and Dilation

The beauty of *Im2Col*() is that it abstracts away how we grab the pixels from the input to form the receptive field for convolution. Padding, stride and dilation all just change the way we grab the pixels and is taken care of by the *Im2Col*() function. Padding tells *Im2Col*() if the pixels are padded one, just put 0 into **B**. Stride tells *Im2Col*() how to move the convolution window around. And dilation tells *Im2Col*() to strech the receiptive field and collect pixels that corresponds to non-zero weights. Since *Col2Im*() is just a reverse process of *Im2Col*(), it will also take padding, stride and dilation into consideration. The core of the computation, i.e. the matrix multiplications, does not change.

Another important influence of padding (P), stride (S) and dilation (D) is that they determine the size of output image, i.e. H_Y and W_Y, which has been missing in the discussion. With some geometry intuition, we can see that

$$H_Y = (H_X + P_T + P_B - (D_H * (H_K - 1) + 1))/S_H + 1 \tag{13}$$

$$W_Y = (W_X + P_L + P_R - (D_W * (W_K - 1) + 1))/S_W + 1 \tag{14}$$

where $P_{\{T,B,L,R\}}$ are paddings from top, bottom, left and right, and $D_{\{H,W\}}$ and $S_{\{H,W\}}$ are dilation and stride along height and width direction.

4.2 Group Filters

[1] [5] introduces group filters to reduce the number of weight so that the computation can fit into the configeration of GPUs. In addition, it seems that group filters can actually improve the accuracy, and the resulting trained filters look more intuitive when plotted as 2D images. For group filters, we partition a set of M filters groups of G, and there will be M/G groups. Each filter in a group, intead of working on all the C channels, works on C/G channels. For all the filters that work on group $g \in \{0, \ldots, M/G-1\}$, we collect them together to form a matrix of size $(M/G) \times ((C/G) \times H_K \times W_K)$.

$$\mathbf{W}_g = [\mathbf{w}^l_{g \times M/G}, \mathbf{w}^l_{g \times M/G+1}, \ldots, \mathbf{w}^l_{(g+1) \times M/G-1}]^T$$

where $\mathbf{w}^l_g$ means a filter g-th group and works on the input channel $[l \times C/G, (l+1) \times C/G - 1]$.

In addition, we need to tell the $Im2Col()$ function to grab pixels from C/G instead of G channels to form the buffer matrix $\mathbf{B}_g$ and its resulting size will be $(C/G \times H_K \times W_K) \times (H_Y \times W_Y)$. Similar to (2), output regarding to this group of filters can be computed as

$$\mathbf{Y}_g = \mathbf{W}_g \mathbf{B}_g \tag{15}$$

Note that $\mathbf{Y}_g \in \mathbb{R}^{(M/G) \times H_Y \times W_Y}$. Repeat this for $g \in \{0, \ldots, M/G-1\}$ and combine the results together and you will get $\mathbf{Y}$.

5 LOCALLY-CONNECTED LAYER

In conventional convolutional layers, the weights of a filters are shared among computation of all the pixels of the output map. In other words, we use this same filter to scan the whole image to extract feature. It makes sense in the case of general images because the objects in interest can appear anywhere in the image. However, for some specific applications such as face recognition, this may not be a good idea, because the input of the face recognition is the frontal face photo where specific features such as eyes and mouth exist in specific region of the image. In this case, we do not want to share the weights of the filters at each location of convolution. We call convolutional layer without weight sharing locally-connected layer.

5.1 Forward Propagation

With out weight sharing, (1) becomes

$$\mathbf{Y}_{m,i,j} = \sum_{c=0}^{C-1} \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \mathbf{W}_{m,c,p,q,i,j} \mathbf{X}_{c,i+p,j+q} + \mathbf{b}_{m,i,j} \tag{16}$$

Notice the additional indices i, j in the weights and biases. Since we have different set of weights for each of the $H_Y \times H_W$ convolutional locations. The input weight tensor is now of size $M \times C \times H_K \times W_K \times H_Y \times W_Y$. Similarly, bias becomes a tenor of size $M \times H_Y \times W_Y$. However, the output $\mathbf{Y}$ is still of the same dimension $(N \times M \times H_Y \times W_Y)$. Again, we will assume $N = 1$ and omit the first dimension in the following discussion.

First, we can still use $Im2Col()$ to take care of the padding, stride and dilation and prepare the buffer matrix $\mathbf{B} \in \mathbb{R}^{(C \times H_K \times W_K) \times (H_Y \times W_Y)}$. However, we cannot do direct matrix multiplication with parts of $\mathbf{W}$ and $\mathbf{B}$ because it will create terms $\mathbf{W}_{m,c,p,q,i,j} \mathbf{X}_{c,i+p,j+q}$ where $i \neq j$. Instead, if we arrange $\mathbf{W}$ into $\mathbf{W}_m \in \mathbb{R}^{(C \times H_K \times W_K) \times (H_Y \times W_Y)}$ for

[1]Group filters currently don't work with dilated convolution in $CAFFE2$

$m = 0, \ldots, M-1$, we can use Hadamard product to compute the first term of the output, ignoring the bias for now:

$$\mathbf{Y}_m = \mathbf{u}^T(\mathbf{W}_m \odot \mathbf{B}) \tag{17}$$

where $\odot$ denotes the Hadamard product (element-wise product) and $\mathbf{u}$ is the same all one vector of size $C \times H_K \times W_K$. The result $\mathbf{Y}_m$ is a $H_Y \times W_Y$ vector. Repeating this for all m and assemble the result, and we will get $\mathbf{Y}$. Some remarks:

(1) When repeating (17) M times, in terms of complexity, it is the same as matrix multiplication in (2). But we are making $2M$ calls (M for Hadamont product and M for AXPY) instead of 1. This may introduce inefficiency compared to the weigth sharing convolutional layer.
(2) cuBLAS and Intel MKL provide support for Hadamond product (DHAD, vdMul) but OpenBLAS doesn't yet [6]. This is not a hard blocker though because it is fairly easy to add this considering OpenBLAS already has elementwise addition.
(3) The AXPY operation with $\mathbf{u}$ in (17) is an overkill. Effectively, we just need a reduction operation. BLAS provides ASUM operation for that. If there is support for that in CUDA, we can just use that.
(4) We can actually convert the Hadamard product to matrix multiplication by creating a huge block-diagnal matrix. BLAS has built-in support for diagnal matrix but I am not sure whether this will help us in terms of performance or not.

Next, we add the bias terms. Since Y and $\mathbf{b}$ are of the size shape. Adding the bias is as simple as a matrix addition

$$\mathbf{Y}' = \mathbf{Y} + \mathbf{b} \tag{18}$$

5.2 Backward Propagation

The input of gradient propagation is the same as normal convulational layer where we have $\mathbf{X}$, $\mathbf{W}$ and $d\mathbf{Y}$, except for that $\mathbf{W}$ is of size $M \times C \times H_K \times W_K \times H_Y \times W_Y$ and $\mathbf{b}$ is of size $M \times H_Y \times W_Y$. For output, $d\mathbf{W}$ will be the same size as $\mathbf{W}$ and bias gradient $d\mathbf{b}$ will be the same size of $\mathbf{b}$. $d\mathbf{X}$ will keep the same size as input $N \times C \times H_X \times H_Y$.

First, we compute the gradient of the weight $\mathbf{W}_{m,c,p,q,i,j}$. Note that for a given filter m, each set of weight for each convolutional operation only contributes to one pixel in the output activation map, instead of $H_Y \times W_Y$ of them. Therefore, we have

$$\frac{\partial E}{\partial \mathbf{W}_{m,c,p,q,i,j}} = \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \frac{\partial \mathbf{Y}_{m,i,j}}{\partial \mathbf{W}_{m,c,p,q,i,j}} = \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \mathbf{X}_{c,i+p,j+q} \tag{19}$$

Notice that compared to (5), it is a single multiplication instead of a convulation. Again, we will iterate through filter $m = 0, \ldots, M-1$ and suppose we already have $\mathbf{B} = \mathit{Im2Col}(\mathbf{X})$. Considering $d\mathbf{Y}$ a $M \times (H_Y \times W_Y)$ matrix, and $\mathbf{B}$ and matrix of size $(C \times H_H \times W_K) \times (H_Y \times W_Y)$,

$$d\mathbf{W}_{m,c,p,q,:,:} = \mathbf{Y}_{m,:} \odot \mathbf{B}_{k,:}$$

computes a vector of $H_Y \times W_Y$ weights, where we first iterate on each row of $\mathbf{Y}$ for $m = 0, \ldots, M-1$ and then on each row of $\mathbf{B}$ for $k = 0, \ldots, C \times H_K \times W_K$. There will be $M \times C \times H_K \times W_K$ Hadamard product of size $H_Y \times W_Y$.

Similarly, the bias gradients can be propagated as

$$\frac{\partial E}{\partial \mathbf{b}_{m,i,j}} = \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \frac{\partial \mathbf{Y}_{m,i,j}}{\partial \mathbf{b}_{m,i,j}} = \frac{\partial E}{\partial \mathbf{Y}_{m,i,j}} \tag{20}$$

So literally, $d\mathbf{b} = d\mathbf{Y}$.

Finally, we compute the input gradient $d\mathbf{X}$. A pixel in input $\mathbf{X}_{c,i,j}$ still contributes to the computation of output activation map $\mathbf{Y}$ in locally connected layer the same way as that in convolutional layer does. However, the weigths are a bit different. The formula to compute $d\mathbf{X}_{c,i,j}$ is as follows

$$\frac{\partial E}{\partial \mathbf{X}_{c,i,j}} = \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \sum_{m=0}^{M-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i-p,j-q}} \frac{\partial \mathbf{Y}_{m,i-p,j-q}}{\partial \mathbf{X}_{c,i,j}} = \sum_{p=0}^{H_K-1} \sum_{q=0}^{W_K-1} \sum_{m=0}^{M-1} \frac{\partial E}{\partial \mathbf{Y}_{m,i-p,j-q}} \mathbf{W}_{m,c,p,q,i,j} \tag{21}$$

The idea is the same as computing $d\mathbf{X}$ in convulational layer. We want to compute a buffer matrix $\mathbf{B}^*$ similar to (11) first and get $d\mathbf{X} = \mathit{Col2Im}(\mathbf{B}^*)$, where each element in $\mathbf{B}^*$ corresponds to $\sum_{m=0}^{M-1} \mathbf{W}_{m,c,p,q,i,j} \mathbf{Y}_{m,i,j}$. Observe each element in $\mathbf{B}^*$ in Figure 2 presents multiplication between $\mathbf{W}$ and $d\mathbf{Y}$ along kernel pixel index as the row index increases and along output pixel index as the column index increases. The difference is that for each column, we have different weights. In order to do that, we first initialze $\mathbf{B}^*$ to be all zero. Next, we iterate on $m = 0, \ldots, M-1$ and arrange $\mathbf{W}$ into M chunks of $\mathbf{W}_m$ (same as in (17)). Then, we iterate on the rows of $\mathbf{W}_m$:

$$\mathbf{B}^*_{k,:} = \mathbf{B}^*_{k,:} + \mathbf{W}_{m_{k,:}} \odot d\mathbf{Y}_{m,:}$$

where $k = 0, \ldots, C \times H_K \times W_K$. Repeating the process by M times and we will obtain the final result of $\mathbf{B}^*$. With that, we can obtain $d\mathbf{X}$ by applying $\mathit{Col2Im}(\mathbf{B}^*)$.

5.3 Efficient implementation with Batched GEMM

From last section, we see that computing the forward and backward propagation usually involves M alls to Hadamard product, which may require multiple kernels calls when implemented on GPU, which is suboptimal. Both Intel MKL and cuBLAS supports an operation called *BatchedGEMM*. *BatchedGEMM* in a simpled form takes two tensor input $A \in \mathbb{R}^{N \times P \times Q}$ and $B \in \mathbb{R}^{N \times Q \times R}$ and produce an output tensor $C \in \mathbb{R}^{N \times P \times R}$, where

$$C_{n,:,:} = A_{n,:,:} B_{n,:,:}$$

It is one single kernel call to the GPU.

To take advantage of that when computing the forward propagation, we now reshape $\mathbf{W}$ to be of size $H_Y \times W_Y \times M \times C \times H_K \times W_K$. We partition it into $H_Y \times W_Y$ matrices $\mathbf{W}_k$ of size $M \times (C \times H_K \times W_K)$ and iterate on k. Each iteration yields M results of $\mathbf{Y}$ on the same convolution location:

$$\mathbf{Y}'_{i,j,:} = \mathbf{W}_k \mathbf{B}_{:,k}$$

Basically, we can implement this with one call to $\mathit{BatchedGEMM}(\mathbf{W}, \mathbf{B}^T)$, where $\mathbf{W}$ is considered a 3D tensor of size $(H_Y \times W_Y) \times M \times (C \times H_K \times W_K)$ and $\mathbf{B}^T$ is considered of size $(H_Y \times W_Y) \times (C \times H_K \times W_K) \times 1$. Notice that $\mathbf{Y}'$ is of size $H_Y \times W_Y \times M$, we might need to reshape it into $\mathbf{Y}$.

For backward propagation of weight gradients, similarly, we will iterate over output pixels $H_Y \times W_Y$ and use that as our batch size to call *BatchedGEMM*. For each $k = 0, \ldots, H_Y \times W_Y - 1$, we take a column of $d\mathbf{Y}$ and a column of $\mathbf{B}$ and computes their outer product, which yields a matrix of $M \times (C \times H_K \times W_K)$ weight gradients $d\mathbf{W}_{i,j,:,:,:,:}$. Implementation is basically one call to $\mathit{BatchedGEMM}(d\mathbf{Y}^T, \mathbf{B})$, where $d\mathbf{Y}^T$ is considered of size $(H_Y \times W_Y) \times M \times 1$.

Finally, for backward propagation of input gradients, we focus on how to compute $\mathbf{B}^*$ with *BatchedGEMM*. Again, we use slices of input weight $\mathbf{W}_k$, for $k = 0, \ldots, H_Y \times W_Y - 1$. In each iteration

$$\mathbf{B}^*_{:,k} = \mathbf{W}_k^T d\mathbf{Y}_{:,k}$$

generates one column of $\mathbf{B}^*$. The whole operation can be wrapped with one call to $BatchedGEMM(\mathbf{W}_k^T, d\mathbf{Y}^T)$. Similarly, resulting $\mathbf{B}^*$ needs reshaping before applying $Col2Im$ to it.

REFERENCES

[1] F.-F. Li, A. Karpathy, and J. Johnson, "CS231n: Convolutional neural networks for visual recognition 2016," 2016. [Online]. Available: http://cs231n.stanford.edu/

[2] A. Gibiansky, "Convolutional neural networks," 2014. [Online]. Available: http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/

[3] J. Kafunah, "Backpropagation in convolutional neural networks," 2016. [Online]. Available: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, p. 2012.

[6] Online, "Hadamard product? issue 1083," 2017. [Online]. Available: https://github.com/xianyi/OpenBLAS/issues/1083