Gradient in convolutional neural networks

1 Fully Connected Layer (affine layer)

Given a D dimensional row vector $\mathbf{x}$ (a single input image), a fully connected layer use the following linear mapping to compute its classification score vector $\mathbf{y}$.

$$\mathbf{y} = \mathbf{x}\mathbf{W} + \mathbf{b} \tag{1}$$

Here $\mathbf{W}$ is a D-by-C weight matrix (C is the number of classes), $\mathbf{b}$ is a C dimensional vector for bias, and $\mathbf{y}$ is the C dimensional output. Further more, in the context of deep convolutional neural networks, let's use $L(\mathbf{y})$ to compute the (classification) loss of this image. For example, L can be the Hinge loss or the Cross entropy loss.

Let's compute the gradient of $\mathbf{y}$ w.r.t $\mathbf{x}$, $\mathbf{W}$ and $\mathbf{b}$; as well as the gradient of loss L w.r.t $\mathbf{x}$, $\mathbf{W}$ and $\mathbf{b}$.

Let's look at $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$. First, what is its dimension? Well, it should be a matrix of size C-by-D. Try to convince yourself with any one of the following three explainations:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \tag{2}$$

- Because $\frac{\partial L}{\partial \mathbf{x}}$ is of size 1-by-D, $\frac{\partial L}{\partial \mathbf{y}}$ is of size 1-by-C, $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ has to be of size C-by-D to make Equation 2 work.
- Because $\mathbf{y}$ has dimension C, and $\mathbf{x}$ has dimension D, so $\frac{\partial L}{\partial \mathbf{y}}$ is of dimension C-by-D (the output dimension by the input dimension)
- We know that $\frac{\partial y_i}{\partial \mathbf{x}}$ should have the same dimension as $\mathbf{x}$ (for y_i is a scaler). It is easy to see from Eq. 1 that

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial \mathbf{x}} \\ \frac{\partial y_2}{\partial \mathbf{x}} \\ \vdots \\ \frac{\partial y_C}{\partial \mathbf{x}} \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_C^T \end{bmatrix} = \mathbf{W}^T \tag{3}$$

Where $\mathbf{w}_i$ is the i-th column (classifier) of the weight matrix $\mathbf{W}$. Apply this to Eq. 3 we have:

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \tag{4}$$

The situation for $\frac{\partial \mathbf{y}}{\partial \mathbf{W}}$ is more interesting. First, what is the dimension of this gradient? Well, we know $\frac{\partial y_i}{\partial \mathbf{W}}$ should be a D-by-C matrix, so $\frac{\partial \mathbf{y}}{\partial \mathbf{W}}$ should be a C-by-D-by-C tensor [1]:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \begin{bmatrix} [\mathbf{x}^T & \mathbf{0} & \dots & \mathbf{0}] \\ [\mathbf{0} & \mathbf{x}^T & \dots & \mathbf{0}] \\ \vdots & \vdots & \ddots & \vdots \\ [\mathbf{0} & \mathbf{0} & \dots & \mathbf{x}^T] \end{bmatrix} \tag{5}$$

[1] Alternatively, you can think the output dimension is C, and the input dimension is D-by-C. So $\frac{\partial \mathbf{y}}{\partial \mathbf{W}}$ is a C-by-D-by-C tensor.

Where each $\begin{bmatrix}\mathbf{x}^T & \mathbf{0} & \dots & \mathbf{0}\end{bmatrix}$ is a D-by-C matrix. Notice in $\frac{\partial \mathbf{y}}{\partial \mathbf{W}}$, the $\mathbf{x}^T$ terms are distributed at the i-th column of the i-th slice (matrix), and all the others entries are zero.

In practice, we do not need to explicitly compute the tensor. Instead, we directly compute $\frac{\partial L}{\partial \mathbf{W}}$ with the following formula:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{W}} = \sum_{i=1}^{C}\frac{\partial L}{\partial \mathbf{y}_i}\frac{\partial \mathbf{y}_i}{\partial \mathbf{W}} = [\frac{\partial L}{\partial y_1}\mathbf{x}^T, \frac{\partial L}{\partial y_2}\mathbf{x}^T, ..., \frac{\partial L}{\partial y_C}\mathbf{x}^T] = \mathbf{x}^T\frac{\partial L}{\partial \mathbf{y}} \tag{6}$$

This means, we only need to compute the multiplication between the transposed data vector and the gradient from the upper layer $\frac{\partial L}{\partial \mathbf{y}}$. Notice $\frac{\partial \mathbf{y}_i}{\partial \mathbf{W}}$ is a D-by-C matrix where the i-th column is $\mathbf{x}^T$, and all other columns are zero.

Similarly, $\frac{\partial \mathbf{y}}{\partial \mathbf{b}}$ is a C-by-C identity matrix,

$$\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \begin{bmatrix}1 & 0 & \dots & 0\\ 0 & 1 & \dots & 0\\ \vdots & \vdots & \ddots & \vdots\\ 0 & 0 & \dots & 1\end{bmatrix} \tag{7}$$

and

$$\frac{\partial L}{\partial \mathbf{b}} = \frac{\partial L}{\partial \mathbf{y}} \tag{8}$$

Question In practice we do not use a single image but a mini-batches of images for training. In this case we can write $\mathbf{X}$ as a data matrix of size N-by-D, where each row is one image. Is Eq. 6 still applicable in this case?

2 Convolutional Layer

Let's start from the forward mapping of convolutional layer, because it is interesting enough for spending a few words. Remember the convolutional operation in deep learning is in fact cross-correlation, which basically computes the dot product between the filter $\mathbf{W}$ and a local neighbourhood of $\mathbf{x}$, and store the output scaler value in $\mathbf{y}$.

Howeve, there is another way to interpret convolution: one can "spray" each value in $\mathbf{x}$ onto $\mathbf{y}$ with $\mathbf{W}'$, which is a 180 degree rotated version of $\mathbf{W}$ (or equivalently, reflected twice – once in the horizontal direction and once in the vertical direction).

Question Use a 1D toy example (with 1-by-three kernel and a 1D input signal) to convince yourself the equivalance of these two interpretations. In this case $\mathbf{W}'$ only needs to be reflected once in the horizontal direction.

Now, let's compute $\frac{\partial L}{\partial \mathbf{x}}$ for a single channel input image $\mathbf{x}$ of M row and N column. Let's assume our convolutional filter has width $2r+1$ (for simple explaination we assume the filter is of odd size). Let's use the chain rule and the fact that $\mathbf{y}$ is a function of $\mathbf{x}$, we have:

$$\frac{\partial L}{\partial \mathbf{x}} = \sum_{i=1}^{M}\sum_{j=1}^{N}\frac{\partial L}{\partial y_{i,j}}\frac{\partial y_{i,j}}{\partial \mathbf{x}} \tag{9}$$

Let's first figure out the dimension of these terms. Since L is a scaler, so $\frac{\partial L}{\partial \mathbf{x}}$ should be the same size of $\mathbf{x}$. The $\frac{\partial L}{\partial y_{i,j}}$ term is also clear – it should be the same size of $y_{i,j}$, which is a scaler. Now let's compute $\frac{\partial y_{i,j}}{\partial \mathbf{x}}$, which should have the same size of $\mathbf{x}$:

$$\frac{\partial y_{i,j}}{\partial \mathbf{x}} = \begin{cases} w_{a-i+r+1,b-j+r+1} & \text{for } x_{a,b} \text{ that } -r <= a-i <= r \text{ and } -r <= b-j <= r\\ 0 & \text{otherwise.}\end{cases} \tag{10}$$

Now, you can imagine that in essence the backward process $\frac{\partial L}{\partial y_{i,j}}\frac{\partial y_{i,j}}{\partial \mathbf{x}}$ "sprays" the value of $\frac{\partial L}{\partial y_{i,j}}$ onto a local neighbourhood around $x_{i,j}$, using $\mathbf{W}$ as the weight. This is almost like the forward process, which sprays each pixel in $\mathbf{x}$ to $\mathbf{y}$ with the rotated filter $\mathbf{W}'$.

Question There is another way to think about it: just like want we saw in the forward case, the spray process is equivalent to a convolutional process with a rotated filter $\mathbf{W}'$. So this means you also do the backward step for a convolutional layer but simply convolve $\frac{\partial L}{\partial \mathbf{y}}$ with $\mathbf{W}'$. Try to convince yourself with a simple 1D example.

Now we know how to backpropogate in convolutional layer. Let's make the problem more interesting. It is often to use stride (larger than 1) in convolution – doing so combines convolution and pooling in a single step. Some people think this even performs better than doing convolution and pooling seperately. The question then is, how can we do backpropogation for strided-convolutional layer?

Question Explain how backpropogation can be implemented for strided-convolution, from both the spray and the convolution with rotated filter perspective. It should be fairly easy from the spray perspective. To do the same job with convolution and the rotated filter, you need to do a simple trick.

Now let's talk about how to compute $\frac{\partial L}{\partial \mathbf{W}}$. It is actually super simple once you know how to compute $\frac{\partial L}{\partial \mathbf{x}}$: just change your perspective and treat $\mathbf{x}$ as a big filter and $\mathbf{W}$ as the input image.

Question Can use a 1D example to illustrate how to compute $\frac{\partial L}{\partial \mathbf{W}}$?

3 ReLU Layer

The ReLU function has the form:

$$y = \max(x, 0) \tag{11}$$

So

$$\frac{\partial y}{\partial x} = x > 0 \ ? \ \frac{\partial L}{\partial y} : 0 \tag{12}$$

.

In practice, this can simply done by multiplying $\frac{\partial L}{\partial \mathbf{y}}$ with a binary mask that computed from $\mathbf{x}$. Notice we can do this for ReLU layer because its input and output has the same size. In the meantime, there is no need to compute $\frac{\partial L}{\partial \mathbf{W}}$ nor $\frac{\partial L}{\partial \mathbf{b}}$ because ReLU layer does not have such parameters.

4 Max Pooling Layer

This is actually interesting. First, there is no parameters in this layer so we only need to compute $\frac{\partial L}{\partial \mathbf{x}}$. Secondly, only the maximum value in the local window survives in the forward pass, so in the backward pass only these values should receive gradient from the upper layer.

Question Can you work out a strategy to compute $\frac{\partial L}{\partial \mathbf{x}}$ for a max pooling layer with filter size 2-by-2 and stride 2?