Backpropagation in Convolutional Networks

Saniya Maheshwari
saniya@firstcry.in

July 16, 2021

Abstract

This is a short note describing how backpropagation is performed in a CNN. In a regular feedforward neural network, the backward pass is straightforward to understand. In a convolutional network, however, we have convolution and max pooling operations which make the backward pass a little complicated. In this note, we discuss how the backward pass through the convolution and max pooling operations can be performed by means of two other operations, namely the transposed convolution and max unpooling operations respectively. We also look at how we can take the gradients of the convolution weights, which is essential for training the network.

1 Max Unpooling

Let us start with the backward pass through the max pooling operation because it is easier to understand. Consider a 4×4 feature map

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Then a max pooling operation (with kernel size 2 and stride 2) identifies the maximal elements (shown in bold) and discards the others:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \to \begin{bmatrix} a_{11} & a_{12} & a_{13} & \mathbf{a_{14}} \\ \mathbf{a_{21}} & a_{22} & a_{23} & a_{24} \\ \mathbf{a_{31}} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & \mathbf{a_{43}} & a_{44} \end{bmatrix} \to \begin{bmatrix} \mathbf{a_{21}} & \mathbf{a_{14}} \\ \mathbf{a_{31}} & \mathbf{a_{43}} \end{bmatrix}$$

Now, in the backward pass, we essentially have a 2×2 matrix of gradients

$$\begin{bmatrix} \frac{\partial F}{\partial a_{21}} & \frac{\partial F}{\partial a_{14}} \\ \frac{\partial F}{\partial a_{31}} & \frac{\partial F}{\partial a_{43}} \end{bmatrix}$$

of a function F with respect to the same maximal elements, and the goal is to somehow “expand” this matrix to get a 4×4 matrix containing the backpropagated gradient. (If we were training the CNN, F would be the loss function – but in general it can be any scalar-valued function of the network.)

The key point is that the non-maximal elements are discarded during the max pooling step, so they play no role in the computation afterward. This means that the gradient of F with respect to these elements must be zero.

So, the backward pass through a max pooling operation essentially reduces to two steps:

1. Move the gradients of the maximal elements to the locations originally occupied by the maximal elements, and
2. fill up the rest of the locations with zeros.

$$\begin{bmatrix} \frac{\partial F}{\partial a_{21}} & \frac{\partial F}{\partial a_{14}} \\ \frac{\partial F}{\partial a_{31}} & \frac{\partial F}{\partial a_{43}} \end{bmatrix} \rightarrow \begin{bmatrix} \cdot & \cdot & \cdot & \frac{\partial F}{\partial a_{14}} \\ \frac{\partial F}{\partial a_{21}} & \cdot & \cdot & \cdot \\ \frac{\partial F}{\partial a_{31}} & \cdot & \cdot & \cdot \\ \cdot & \cdot & \frac{\partial F}{\partial a_{43}} & \cdot \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & \frac{\partial F}{\partial a_{14}} \\ \frac{\partial F}{\partial a_{21}} & 0 & 0 & 0 \\ \frac{\partial F}{\partial a_{31}} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial F}{\partial a_{43}} & 0 \end{bmatrix}$$

That’s it!

This operation is known as *max unpooling.* It can also be used in another sense – if we were to “unpool” the maximal values themselves, instead of their gradients:

$$\begin{bmatrix} \mathbf{a_{21}} & \mathbf{a_{14}} \\ \mathbf{a_{31}} & \mathbf{a_{43}} \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & \mathbf{a_{41}} \\ \mathbf{a_{21}} & 0 & 0 & 0 \\ \mathbf{a_{31}} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{a_{43}} & 0 \end{bmatrix}$$

Here, as you can see, the maximal values have been restored to their original locations, while the non-maximal values have been replaced by zeros. In this way, max unpooling can be viewed as computing a *partial inverse* of the max pooling operation [8].

One thing to note is that, in order to perform a max unpooling operation, we have to keep track of the locations of the maximal elements during the forward pass through the max pooling operation. These locations are sometimes known as *switches* [8].

2 Transposed Convolutions

Consider a convolution operation involving a 4×4 feature map, represented by the matrix of a_{ij}’s, and a 3×3 kernel represented by the matrix of k_{ij}’s. The convolution between these two is denoted by $*$ and the stride used is 1.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \tag{1}$$

The result of this convolution is a 2×2 feature map:

$$\begin{bmatrix} \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{ij} & \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i,j+1} \\ \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i+1,j} & \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i+1,j+1} \end{bmatrix}$$

Now, suppose we were to flatten out the 4×4 input feature map into a 16-dimensional vector. Then, realize that the following matrix multiplication

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & 0 & k_{11} & k_{12} & k_{13} & 0 & \dots & 0 \\ 0 & k_{11} & k_{12} & k_{13} & 0 & k_{11} & k_{12} & k_{13} & \dots & 0 \\ 0 & 0 & 0 & 0 & k_{11} & k_{12} & k_{13} & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \dots & k_{33} \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{14} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{24} \\ \vdots \\ a_{44} \end{bmatrix}$$

is identical to the 2×2 feature map we found above, flattened into a 4-dimensional vector.

What this shows us is that a convolution operation can be expressed as a regular linear multiplication operation of a feedforward neural network. If we flatten out the input and output feature maps, the convolution kernel can be written out as a weight matrix [1]. This weight matrix will have a lot of zeros and will repeat the kernel weights throughout. Indeed – it is a well-known fact that a CNN is equivalent to a feedforward neural network, with two additional properties: *sparse connections* and *weight sharing* [2].

Then, computing the backward pass through a convolution operation is simply a matter of applying the regular chain rule. If h and a denote the flattened output and input feature maps respectively, the forward pass is

$$h = Wa$$

and the backward pass is

$$\frac{\partial F}{\partial a} = W^T \left(\frac{\partial F}{\partial h} \right) \tag{2}$$

Remember that W is a weight matrix derived from the kernel weights. The backward pass uses the transpose of the weight matrix, hence this operation is known as a *transposed convolution.*

Now, let us go ahead and expand the right-hand side of the rule (2).

$$\begin{bmatrix} k_{11} & 0 & 0 & 0 \\ k_{12} & k_{11} & 0 & 0 \\ k_{13} & k_{12} & 0 & 0 \\ 0 & k_{13} & 0 & 0 \\ k_{11} & 0 & k_{11} & 0 \\ k_{12} & k_{11} & k_{12} & 0 \\ k_{13} & k_{12} & k_{13} & 0 \\ 0 & k_{13} & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & k_{33} \end{bmatrix} \begin{bmatrix} \frac{\partial F}{\partial h_{11}} \\ \frac{\partial F}{\partial h_{12}} \\ \frac{\partial F}{\partial h_{21}} \\ \frac{\partial F}{\partial h_{22}} \end{bmatrix}$$

If you observe carefully, the result of this matrix multiplication, when "un-flattened", is identical to the result of the following convolution operation (with stride 1):

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial F}{\partial h_{11}} & \frac{\partial F}{\partial h_{12}} & 0 & 0 \\ 0 & 0 & \frac{\partial F}{\partial h_{21}} & \frac{\partial F}{\partial h_{22}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} k_{33} & k_{32} & k_{31} \\ k_{23} & k_{22} & k_{21} \\ k_{13} & k_{12} & k_{11} \end{bmatrix} \tag{3}$$

The feature map on the left is the gradient matrix that arrives as input during the backward pass, with a padding of 2. On the right is a kernel which is identical to the original convolution kernel in (1), except that it is *flipped* about the horizontal and vertical axes.

This means that the backward pass through a convolution operation can be performed using another convolution operation – which utilizes sufficient padding and a kernel formed by flipping the kernel of the original convolution operation horizontally and vertically [8, 3]. (This explains the "convolution" part of the name "transposed convolution".)

(Do note that it is the weight matrix derived from the convolution kernel that is transposed, and not the convolution kernel itself. The kernel is flipped horizontally and vertically, which is not the same as transposing.)

Recall that the stride used in the original convolution operation (1) was 1. If we had used a different stride, it turns out that the corresponding transposed convolution would be slightly different from the one in (3). The kernel would be flipped in the same way, but along with adding zeros on the boundary of the feature map as padding, we would also have to add zeros in between the elements of the feature map [1]. For example, consider the following convolution operation with a stride of 2:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \tag{4}$$

This will produce a 2×2 feature map:

$$\begin{bmatrix} \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{ij} & \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i,j+2} \\ \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i+2,j} & \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i+2,j+2} \end{bmatrix}$$

Then the corresponding transposed convolution takes the form

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial F}{\partial h_{11}} & 0 & \frac{\partial F}{\partial h_{12}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial F}{\partial h_{21}} & 0 & \frac{\partial F}{\partial h_{22}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} k_{33} & k_{32} & k_{31} \\ k_{23} & k_{22} & k_{21} \\ k_{13} & k_{12} & k_{11} \end{bmatrix}$$

As you can see, one row/column of zeros has been added between the elements of the gradient matrix.

Such a convolution operation, which has zeros in between the elements of the input feature map, can be viewed as having a stride less than 1 (though while hand-doing the convolution we use a stride of 1), because the kernel moves over the feature map at a slower pace than before. For this reason, transposed convolutions are also known as *fractionally-strided* convolutions [1].

Another name that is often given to the transposed convolution operation is that of a *deconvolution*. However, this is a misnomer [1] because the transposed convolution does nothing to invert the convolution operation – it uses W^T, not W^{-1}. At most, this operation only inverts the shape of a convolution operation, i.e. if a convolution operation produces 2×2 feature maps given 4×4 feature maps, the corresponding transposed convolution operation produces 4×4 feature maps given 2×2 feature maps.

3 Convolution Weights

We have seen how to take gradients through a convolution operation but we still have to take gradients with respect to the parameters of the convolution operation, i.e. the kernel weights. Let us discuss this next.

Consider again the convolution operation (1). Its output is a 2×2 feature map which we shall denote as

$$\begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$$

Take one of the elements in this feature map, say h_{12}. Clearly

$$h_{12} = \sum_{i=1}^{3} \sum_{j=1}^{3} k_{ij} a_{i,j+1}$$

Now fix a particular k_{ij}. Then

$$\frac{\partial h_{12}}{\partial k_{ij}} = a_{i,j+1}$$

In a similar manner, this particular k_{ij} influences all the other elements in the feature map and not just h_{12}.

$$\frac{\partial h_{i'j'}}{\partial k_{ij}} = a_{i+i'-1,j+j'-1}$$

Then, by the chain rule, the expression for the gradient of F with respect to k_{ij} would be

$$\frac{\partial F}{\partial k_{ij}} = \sum_{i'=1}^{2} \sum_{j'=1}^{2} \frac{\partial F}{\partial h_{i'j'}} \frac{\partial h_{i'j'}}{\partial k_{ij}}$$

or

$$\frac{\partial F}{\partial k_{ij}} = \sum_{i'=1}^{2} \sum_{j'=1}^{2} \left(\frac{\partial F}{\partial h_{i'j'}} \right) a_{i+i'-1,j+j'-1}$$

To better understand what this expression means, let us substitute some values of i and j. First take $i = j = 1$, i.e. we are looking at the gradient of k_{11}:

$$\frac{\partial F}{\partial k_{11}} = \sum_{i'=1}^{2} \sum_{j'=1}^{2} \left(\frac{\partial F}{\partial h_{i'j'}} \right) a_{i'j'}$$

Then take $i = 1$, $j = 2$:

$$\frac{\partial F}{\partial k_{12}} = \sum_{i'=1}^{2} \sum_{j'=1}^{2} \left(\frac{\partial F}{\partial h_{i'j'}} \right) a_{i',j'+1}$$

You should see a familiar pattern emerging. Indeed, the gradients with respect to the kernel weights can be obtained using the following convolution operation [3]:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial h_{11}} & \frac{\partial F}{\partial h_{12}} \\ \frac{\partial F}{\partial h_{21}} & \frac{\partial F}{\partial h_{22}} \end{bmatrix}$$

where the feature map on the left is the input to the convolution operation during the forward pass, and the kernel on the right is the gradient matrix that arrives as input during the backward pass. Neat!

What if the stride of the original convolution is greater than 1? The same result holds, except that we would have to introduce some *dilation*. For example, if the original convolution was of the form of (4) with a stride of 2, then the convolution yielding the gradients of the kernel weights would be

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix} * \begin{bmatrix} \frac{\partial F}{\partial h_{11}} & \frac{\partial F}{\partial h_{12}} \\ \frac{\partial F}{\partial h_{21}} & \frac{\partial F}{\partial h_{22}} \end{bmatrix}$$

with a stride of 1 but a dilation of 2.

4 Applications

Transposed convolution and max unpooling operations are used behind-the-scenes in deep learning libraries to compute the backward pass through a CNN [7].

They have found explicit use in architectures known as *deconvolutional networks*. In [8], these operations are used to perform a modified backward pass [6] through the convolutional layers of AlexNet [4]. Other examples include [9] which aims to reconstruct images passed as input to the network, and [5] which performs semantic image segmentation.

References

[1] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *arXiv preprint arXiv:1603.07285* (2016).

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[3] Jefkine Kafunah. *Backpropagation in Convolutional Neural Networks*. Accessed: 2021-07-14. URL: `https://jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/`.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[5] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.

[6] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep inside convolutional networks: Visualising image classification models and saliency maps". In: *arXiv preprint arXiv:1312.6034* (2013).

[7] PyTorch Team. *torch/nn/grad.py*. Accessed: 2021-07-14. URL: `https://github.com/pytorch/pytorch/blob/master/torch/nn/grad.py#L165`.

[8] Matthew D Zeiler and Rob Fergus. "Visualizing and understanding convolutional networks". In: *European conference on computer vision*. Springer. 2014, pp. 818–833.

[9] Matthew D Zeiler et al. "Deconvolutional networks". In: *2010 IEEE Computer Society Conference on computer vision and pattern recognition*. IEEE. 2010, pp. 2528–2535.