Chpt8 Using convolutions to generalize

Sep 5, 2021

Covers:

1. Understanding convolution
2. Building a convolutional neural network
3. Creating custom nn.Module subclasses
4. Design choices for neural network

Due to the fully connected setup needed to detect the various possible translations of the bird or airplane in the image, we have both too many parameters and no position independence. As discussed in the last chapter, we could argument our training data by using a wide variety of recropped images to try to force generalization, but that won't address the issue of having too many parameters. There is a better way. It consists of replacing the dense, fully connected affine transformation in our neural network unit with a different linear operation: convolution.

8.1 The case for convolutions

In this section, we'll see how convolutions deliver locality and translation invariance. We could compute the weighted sum of a pixel with its intermediate neighbors. This would be equivalent to building weight matrices, one per output feature and output pixel location, in which all weights beyond the certain distance are zero. This will still be a weighted sum: that is, a linear operation.

We would like these localized patterns to have an effect on the output regardless of their location in the image: that is, to be **translation invariant**.

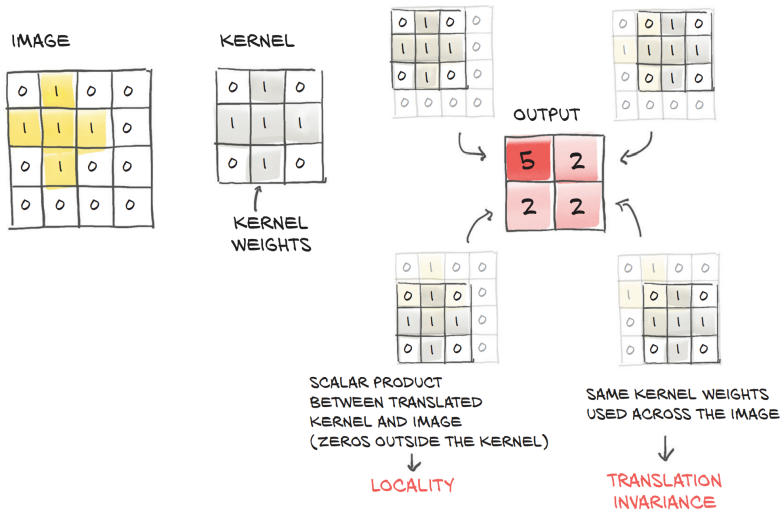

Figure 8.1 Convolution: locality and translation invariance

A convoultion is equivalent to having multiple linear operations whose weights are zero almost everywhere except around individual pixels and that receive equal updates during training.

Summarizing, by switching to convolutions, we get:

1. Local operations on neighbourhoods
2. Translation invariance
3. Model with a lot fewer parameters

The key insight underlying the third point is that, with a convolution layer, the number of parameters depends on the size of convolution kernel (3×3, 5×5, and so on) and how many convolution filters we decide to use in our model.

8.2 Convolutions in action

8.2.1 Padding the boundary

The fact that our output image is smaller than the input is side effect of deciding what to do at the boundary of the image. Pytorch gives us the possibility of *padding* the image by creating *ghost* pixel around the border that have value zero as far as the convolution is concerned.

In our case, specifying padding=1, when kernel_size=3 means i00 has an extra set of neighbors above it and to its left, so that an output of the convolution can be computed even in the corner of our original image. The net result is that the output has now the exact same size as the input.

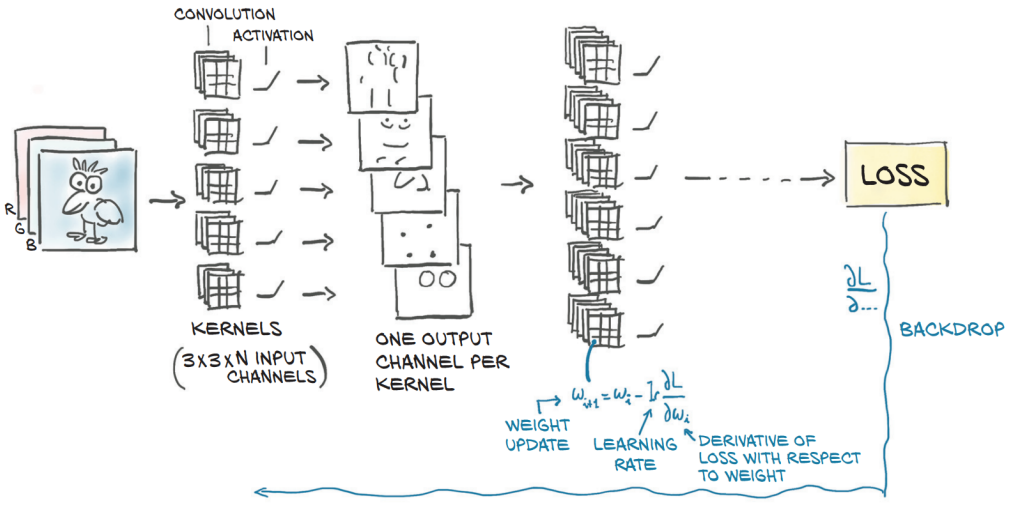

Figure 8.6 The process of learning with convolutions by estimating the gradient at the kernel weights and updating them individually in order to optimize for the loss

8.2.3 Looking further with depth and pooling

Another option, which is used in convolutional neural networks, is stacking one convolution after the other and at the same time downsampling the image between successive convolutions.

How we compute the value of the output based on the values of the input is up to us. We could:

1. Average the four pixels.
2. Take the maximum of the four pixels, called *max pooling*.

3. Performing a strided convolution, where only every Nth pixel is calculated. A 3×4 convlution with stride 2 still incorporates input from all pixels from the previous layer.

We will be focusing on max pooling. The output images from a convolution layer, especially since they are followed by an activation just like any other linear layer, tend to have a high magnitude where certain features corresponding to the estimated kernel are detected.

Combining convolutions and downsampling for great good

In figure, we start by applying a set of 3×3 kernels on our 8×8 image, obtaining a multichannel output image of the same size. Then we scale down the output image by half, obtaining a 4×4 image, and apply another set of 3×3 kernels to it. This second set of kernels operates on a 3×3 neighborhood of something that has been scaled down by half, so it effectively maps back to 8×8 neighborhoods of the input. In addition, the second set of kernels takes the output of the first set of kernels and extracts additional features on top of those.

So, on one hand, the first set of kernels operates on small neighborhoods on first-order, low-level features, while the second set of kernels effectively operates on wider neighborhoods, producing features that are compositions of the previous features. This is a very powerful mechanism that provides convolutional neural networks with the ability to see into very complex scenes-much more complex than our 32×32 images

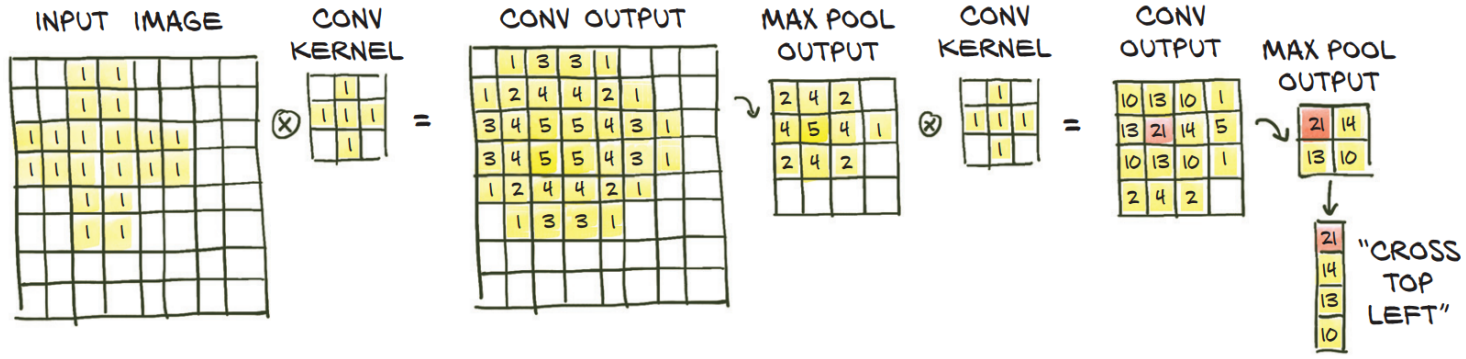

Figure 8.8 More convolutions by hand, showing the effect of stacking convolutions and downsampling: a large cross is highlighted using two small, cross-shaped kernels and max pooling.