

Convolutional Neural Networks: The Convolution

Kevin Fu*

January 2019

1 Introduction

Convolutional neural networks (CNN) are variants on standard neural networks that are designed to take advantage of spatial relationships. They're currently the industry standard for image recognition tasks—the top performers on the MNIST dataset, with error rates from 0.31 to 0.27, are all CNNs.

2 Convolutional Layers

In theory, a standard one-layer neural network can learn any function, provided the single layer is large enough. However, in practice, we often add layers to shorten training times and decrease the chance of overfitting. Likewise, to improve performance on problems like image recognition, where spatial relationships are important, we modify the structure of a vanilla neural network. But rather than simply adding layers, we change the structure of the layers themselves.

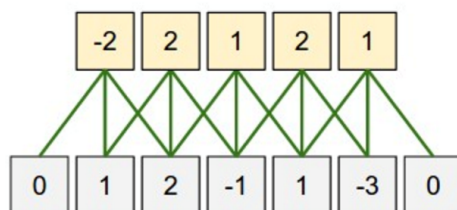


Figure 1: A fully-connected layer.

Traditional neural networks are fully connected, meaning every neuron in one layer is connected to every other neuron in the subsequent layer, as shown in Figure 1. This means that when we take images and flatten them into a

*Expansion of Sylesh Suresh and Mihir Patel's lectures of the same name

vector, as you did with the MNIST images for the Neural Network competition, we're disregarding the fact that pixels close together are more correlated than pixels on opposite sides of the image. The neural network has to learn these relationships for itself.

This works for smaller images, like the 28x28 images in the MNIST dataset, but larger images contain hundreds of thousands of pixels, which increases the time it takes our network to learn spatial relationships. It would be easier if we structured our neural network to correlate nearby pixels, without forcing it to backpropagate for that understanding.

2.1 Convolutions

To take advantage of spatial relationships, CNNs employ convolutional layers, so named because they rely on *convolutions*. In computer vision, a convolution is an operation of a *kernel* on an image:

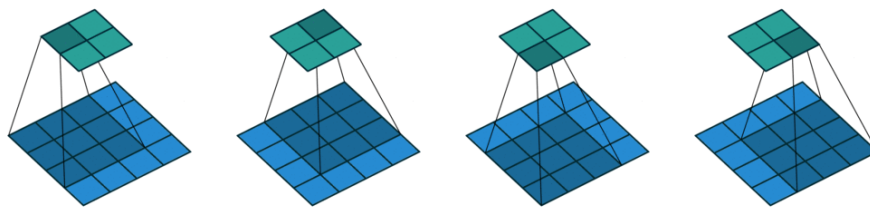
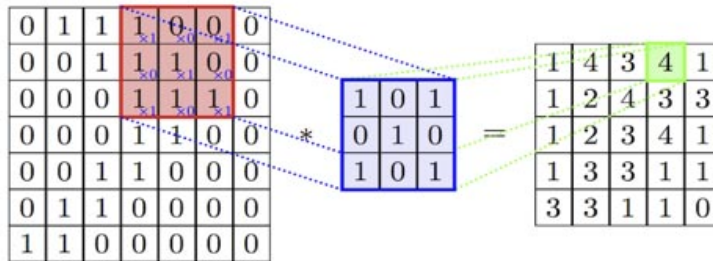


Figure 2: A 3x3 convolution.

These diagrams show grids, but internally, all the images, kernels, and outputs are matrices of floats. Grayscale images are matrices of light intensity levels, so each pixel maps with one number. Color images use three matrices, each with red, green, or blue intensity levels; hence, three numbers per pixel.

In Figure 2, the bottom grid is the input image, the top grid is the output, and the gray shadow is the kernel that's slid over the input to get each square of the output. The 3x3 kernel is slid left-to-right, top-to-bottom, over the whole image. At each step, the pixel intensities of the kernel-covered region of the image are multiplied with the corresponding values of the kernel, then added together to produce a single number for the output matrix.



You'll notice that the output grid is one pixel smaller than the input grid above, due to the nature of sliding the kernel around. To make the dimensions match, we can add a border of zeroes, known as *zero-padding*:

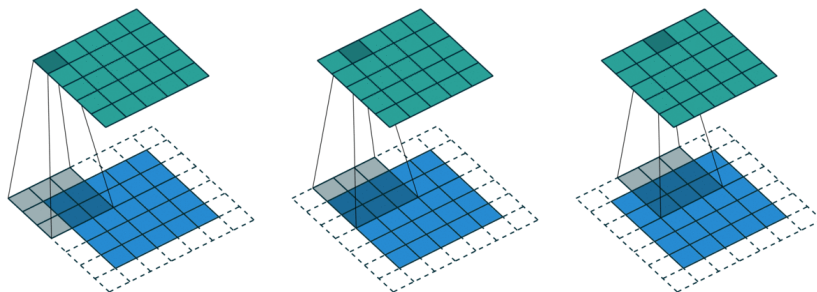


Figure 3: A 3x3 convolution with zero-padding.

Consider a convolution on a zero-padded image with this kernel (sometimes called a filter or a mask):

$$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This convolution's output is a blurry version of the input image:

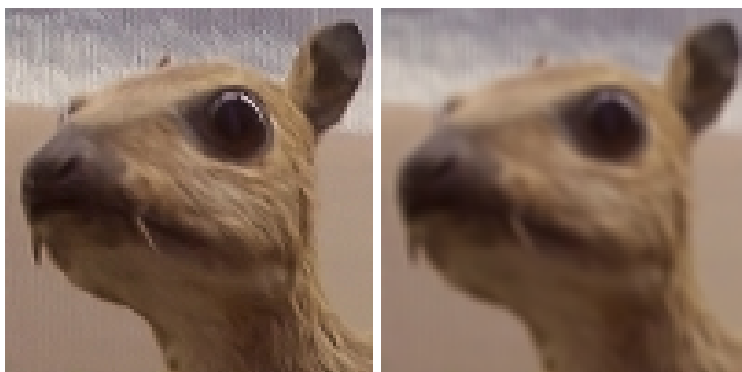


Figure 4: Image blurred with box blur kernel.

This makes sense if you look at the kernel (known as a box blur). A kernel of ones means the output pixels are simply the sum of the input pixel and its eight neighbors, and then we divide by nine to normalize. Keep the intuition that kernels and convolutions relate a pixel to its neighbors in mind.

A more complex example: convolving the kernel in Figure 5, known as a Sobel kernel, is a simple way to detect vertical edges in an image:

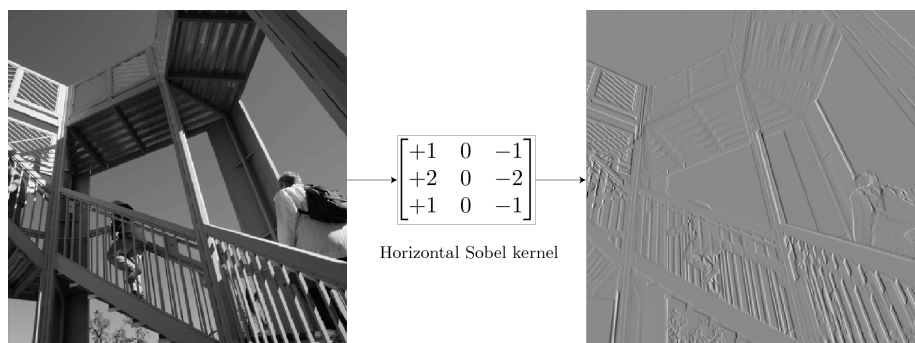


Figure 5: Vertical edges found with a Sobel kernel.

For areas of the image that are the same color, this kernel will output zero, as the left column and right column cancel each other out. But areas with vertical edges have one color on the left side of the edge and a different color on the right side. Hence, the Sobel kernel will output high values over those regions.

2.2 Feature Maps

For a CNN, we treat the values of our kernel as weights, and have our forward propagation apply convolutions on input images rather than the traditional fully-connected layers. That way, our neural network is forced to focus on a local region of pixels, rather than guess at which pixels are correlated across the entire image. (You may see this described as each neuron having a “local receptive field.”)

Like in a normal kernel, the weights of our CNN’s kernel are the same across the whole image. This makes sense given what we know about convolutions, but it’s hard to wrap your head around if you consider it from a neural network perspective: how could a layer with only *nine* weights possibly be useful on an image, which has thousands of input values?

In reality, CNNs use multiple kernels in each convolutional layer. Each kernel produces what’s called a *feature map*, so named because the CNN will hopefully use each kernel to find different features in an image. For example, this kernel:

$$\begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$$

is good at finding parts of an image that look like it. See Figure 6.

The output feature map, produced by convolving the kernel over the input image, has higher values where there are diagonals in the input image. We can add as many kernels as we want to extract multiple feature maps from the same image, each adding a level of understanding to our network.

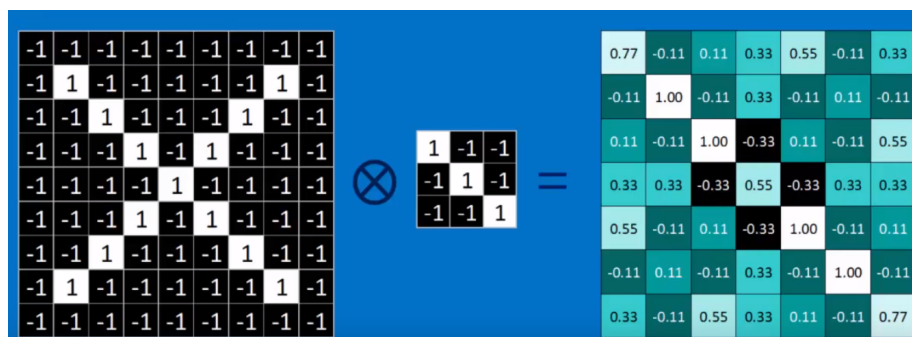


Figure 6: Finding diagonals in an image.

2.3 Multiple Channels

Notice that feature maps, or any of the outputs to convolutions above, look awfully similar to the input images we fed them, especially if we use zero-padding to keep the dimensions consistent. Logically, we can treat feature maps as the inputs to further layers in our CNN, because convolving over a feature map is the same as convolving over an input image. Both are stored as matrices of floats.

Earlier, we noted that CNNs use multiple convolutions on the same input image to produce multiple feature maps, and thus extract multiple features of an image. This is how we handle the first layer of a grayscale image. But this poses a question: sure, we can treat an output feature map as the input to another convolution, but how do we convolve over multiple feature maps? If you saw Figure 4 and wondered how we applied a single kernel to a color image, which we know is actually three different matrices, this question may have come up for you already.

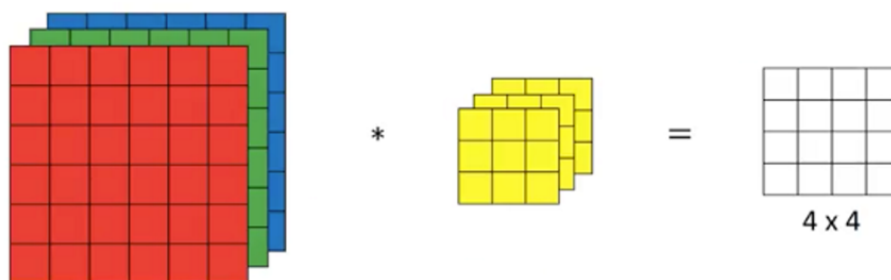


Figure 7: A multi-channel convolution.

Think of multiple correlated feature maps (or multiple color matrices of an RGB image) as a *stack*. The depth of this stack is however many kernels we ran on the previous layer, because each convolution produces a 2D output. Each

layer of our stack is called a *channel*: a grayscale image has one channel and an RGB image has three. To run a convolution over a multi-channel input, we convolve a unique 2D kernel across every 2D channel of the input, then sum each kernel's feature map for its respective channel. This results in one final 2D feature map for a multi-channel input. See Figure 7.¹

Just like the input stack, this group of kernels-per-input-channel can be thought of as a *stack* of kernels. Together, this stack makes up a single *filter*, as shown in Figure 8. (Note: many sources on CNNs will use kernels and filters interchangeably, because they are the same thing—in the 2D case. However, the distinction is clear for multiple channels: kernels are 2D-only, whereas filters can have multiple channels.)

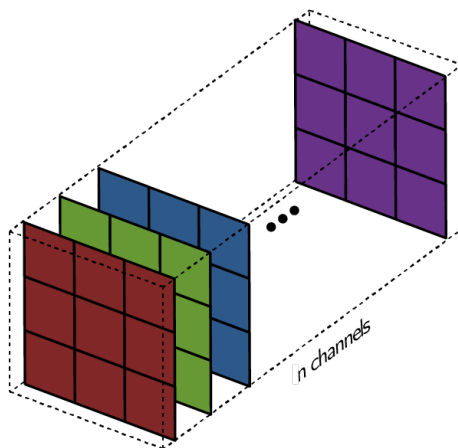


Figure 8: A filter, which is an n -channel stack of kernels.

So adding subsequent layers to our CNN is the same as the transition from a one-channel grayscale image to a multi-channel stack of feature maps, if we think of a single kernel as a one-channel filter. Applying one filter to the input, if the channels between the filter and the input match, will yield a 2D feature map. Applying multiple filters will allow us to detect multiple features in the input, and give us multiple 2D feature maps. We can stack these feature maps together and feed it to yet another convolutional layer.

In sum, a convolutional layer is not a single kernel passed over a single input channel, but as many kernels as desired passed over as many input channels as given. For a layer with n kernels, the resulting stack of feature maps will have n -channels. This allows us to handle three-channel color images as direct inputs to our CNNs.

¹For an animated visualization of this, go to <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>.

2.4 Comparing CNNs to Vanilla Neural Nets

The initial reason we chose convolutional layers over fully-connected ones was to take advantage of spatial structuring in an image—the idea that pixels close together are more correlated than pixels far apart. However, by using convolutions, we’ve also solved another problem: dealing with larger input images. A vanilla neural network was fine for the MNIST dataset, with its 28x28x1 inputs, but moving to a modest 224x224x3 means we need a input layer of 150,000 neurons. Attempting to add a single hidden layer to halve that input would require over 10 billion parameters (weights and biases).

Having weighted kernels, which can be thought of as locally-connected neurons, dramatically reduces the number of weights needed from layer to layer. For example, the record-setting 152-layer deep ResNet architecture, which runs on 224x224x3 input images, has less than 20 million parameters *total*. Of course, they employed a few more tricks to reduce parameters and thus reduce training times.

Next week, in the second part of this lecture, we’ll learn how stacks of feature maps are implemented as convolutional layers, how those stacks are turned into predictions, and improvements on the basic CNN framework.

3 References

3.1 Information Sources

- https://en.wikipedia.org/wiki/MNIST_database
- <https://www.youtube.com/watch?v=FmpDIaiMIeA>
- <http://neuralnetworksanddeeplearning.com/chap1.html>
- <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>
- <https://arxiv.org/pdf/1707.09725.pdf#page=17>
- <https://ai.stackexchange.com/questions/5769/in-a-cnn-does-each-new-filter-have-different-weights-for-each-input-channel-or>
- 2018 and 2019 TJML CNN Lectures

3.2 Image Sources

- https://github.com/vdumoulin/conv_arithmetic
- <https://www.saama.com/blog/different-kinds-convolutional-filters/>
- [https://en.wikipedia.org/wiki/Kernel_\(image_processing\)#Convolution](https://en.wikipedia.org/wiki/Kernel_(image_processing)#Convolution)
- <https://towardsdatascience.com/intuitively-understanding-convolutions-for-deep-learning-1f6f42faee1>