Lecture 9: Convolutional Networks

Roger Grosse

1 Introduction

So far, all the neural networks we've looked at consisted of layers which computed a linear function followed by a nonlinearity:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x}). \tag{1}$$

We never gave these layers a name, since they're the only thing we used. Now we will. They're called **fully connected** layers, because every one of the input units is connected to every one of the output units. While fully connected layers are useful, they're not always what we want. Here are some reasons:

- They require a lot of connections: if the input layer has M units and the output layer has N units, then we need MN connections. This can be quite a lot; for instance, suppose the input layer is an image consisting of $M = 256 \times 256 = 65563$ grayscale pixels, and the output layer consists of $N = 1000$ units (modest by today's standards). A fully connected layer would require 65 million connections. This causes two problems:
  - Computing the hidden activations requires one add-multiply operation per connection in the network, so large numbers of connections can be expensive.
  - Each connection has a separate weight parameter, so we would need a huge number of training examples in order to avoid overfitting.
- If we're trying to classify an image or an audio waveform, there's certain structure we'd like to make use of. For instance, features (such as edges) which are useful at one image location are likely to be useful at other locations as well. We would like to **share structure** between different parts of the network. Another property we'd like to make use of is **invariance**: if the image or waveform is transformed slightly (e.g. by shifting it a few pixels), the classification shouldn't change. Both of these properties should be encoded into the network's architecture if possible.

For the next three lectures, we'll talk about a particular kind of network architecture which deals with all these issues: the **convolutional network**, or **conv net** for short. Like the name suggests, the architecture is inspired by a mathematical operator called **convolution** (which we'll explain shortly).

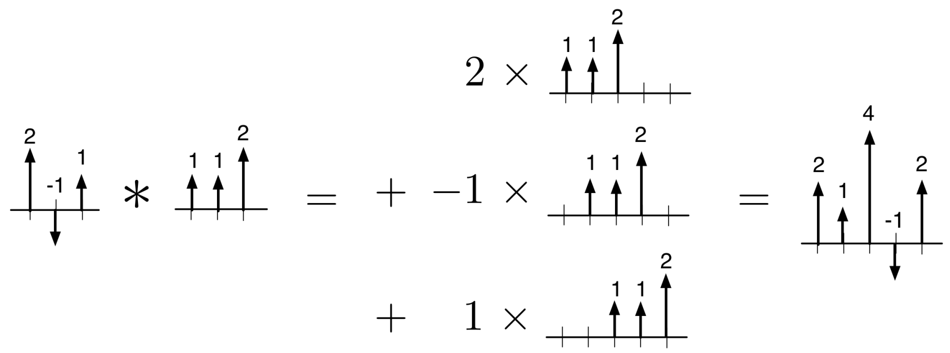


Figure 1: Translate-and-scale interpretation of convolution of one-dimensional signals.

Conv nets revolutionized the field of computer vision in 2012, and by now, the vast majority of papers published in top computer vision conferences use conv nets in some way. Fortunately, the ideas aren't terribly complicated, and by the end of these three lectures, you'll understand how these things work. With a relatively small number of lines of code in a framework like PyTorch or TensorFlow, you can build a computer vision system more powerful than the state-of-the-art just a few years ago.

2 Convolution

Before we talk about conv nets, let's introduce convolution. Suppose we have two **signals** x and w, which you can think of as arrays, with elements denoted as $x[t]$ and so on. As you can guess based on the letters, you can think of x as an input signal (such as a waveform or an image) and w as a set of weights, which we'll refer to as a **filter** or **kernel**. Normally the signals we work with are finite in extent, but it is sometimes convenient to treat them as infinitely large by treating the values as zero everywhere else; this is known as **zero padding**.

Let's start with the one-dimensional case. The **convolution** of x and w, denoted $x * w$, is a signal with entries given by

$$(x * w)[t] = \sum_{\tau} x[t - \tau]\, w[\tau]. \tag{2}$$

There are two ways to think about this equation. The first is **translate-and-scale**: the signal $x * w$ is composed of multiple copies of x, translated and scaled by various amounts according to the entries of w. An example of this is shown in Figure 1.

A second way to think about it is **flip-and-filter**. Here we generate each of the entries of $x * w$ by flipping w, shifting it, and taking the dot product with x. An example is shown in Figure 2.

The two-dimensional case is exactly analogous to the one-dimensional case; we apply the same definition, but with more indices:

$$(x * w)[s, t] = \sum_{\sigma,\tau} x[s - \sigma, t - \tau]\, w[\sigma, \tau]. \tag{3}$$

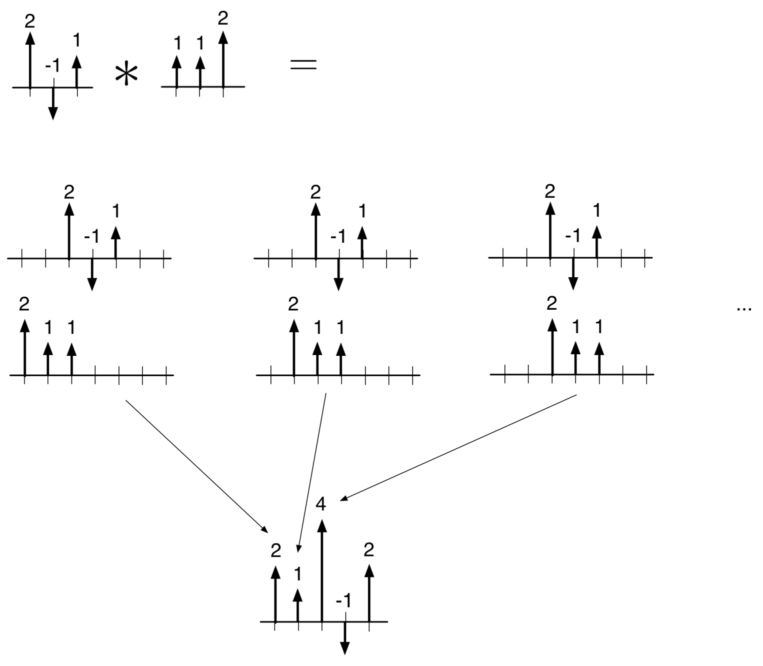


Figure 2: Flip-and-filter interpretation of convolution of one-dimensional signals.

This is shown graphically in Figures 3 and 4.

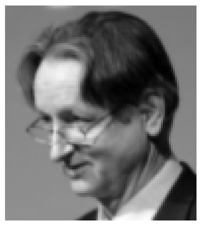
2.1 Examples

Despite the simplicity of the operation, convolution can do some pretty interesting things. For instance, we can blur an image:

$*$

0	1	0
1	4	1
0	1	0

$=$

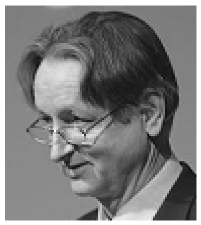


We can sharpen it:

$*$

0	-1	0
-1	8	-1
0	-1	0

$=$



If we change the values slightly, we get a very different effect. (Why? What is the difference from the previous example?) This is a center-surround filter, and it responds only to boundaries.

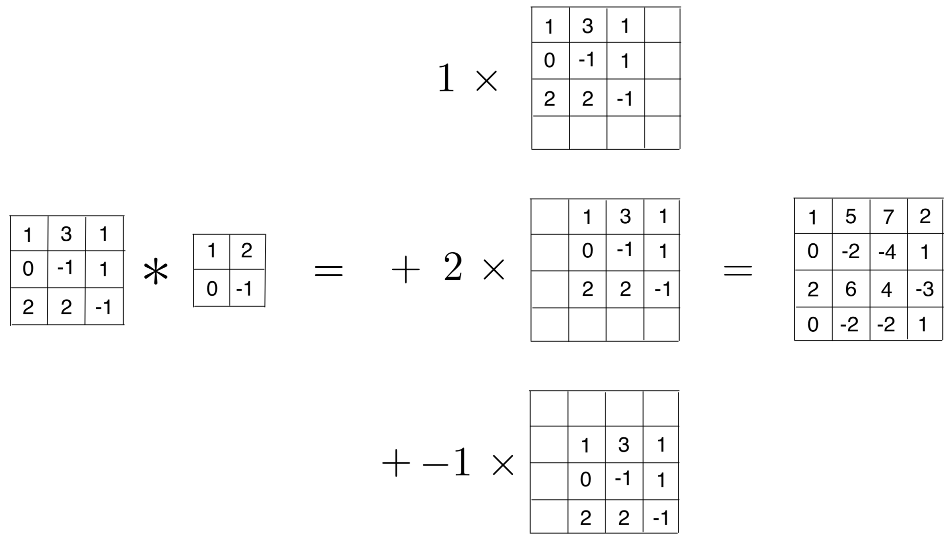


Figure 3: Translate-and-scale interpretation of convolution of two-dimensional signals.

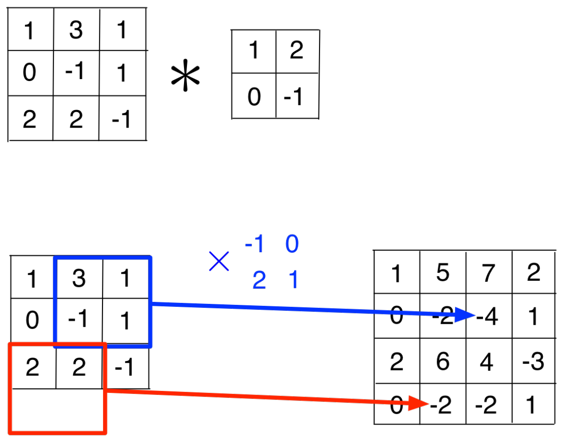
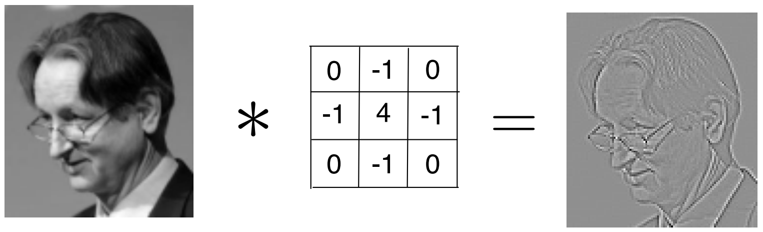
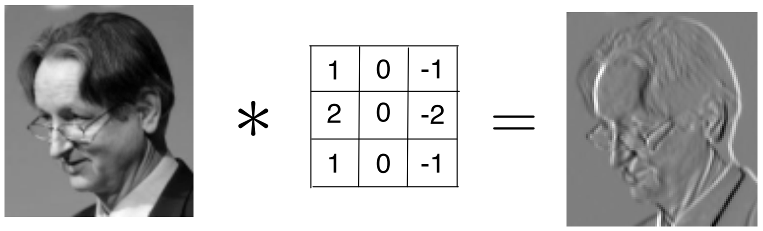


Figure 4: Flip-and-filter interpretation of convolution of two-dimensional signals.



We can detect edges. (That is, edges in the image itself, rather than edges in the world. Detecting edges in the world is a very hard problem.) This filter is known as a Sobel filter.



2.2 Properties of convolution

Now that we've seen some examples of convolution, let's note some useful properties. First of all, it behaves like multiplication, in that it's commutative and associative:

It's a good exercise to verify both properties from the definition.

$$u * v = v * u \tag{4}$$
$$(u * v) * w = u * (v * w). \tag{5}$$

While both properties follow easily from the definition, they're a bit surprising and counterintuitive when you think about flip-and-filter. For instance, let's say you blur the image and then run a horizontal edge filter, represented as $(x * w_{\text{blur}}) * w_{\text{horz}}$. By commutativity and associativity, this is equivalent to first running the edge filter, and then blurring the result, i.e. $(x * w_{\text{horz}}) * w_{\text{blur}}$. It's also equivalent to convolving the image with a single kernel which is obtained by blurring the edge kernel: $x * (w_{\text{horz}} * w_{\text{blur}})$.

Another useful property of convolution is that it is **linear**:

$$(ax + bx') * w = ax * w + bx' * w \tag{6}$$
$$x * (aw + bw') = ax * w + bx * w'. \tag{7}$$

This is convenient, because linear operations are often easier to deal with. But it also shows an inherent limit to convolution: if you have a neural net which computes lots of convolutions in sequence, it can still only compute linear functions. In order to compute more complex operations, we'll need to apply some sort of nonlinear activation function in each layer. (More on this later.)

One last property of convolution is that it's **equivariant** to translation. This means that if we shift, or translate, x by some amount, then the output $x * w$ is shifted by the same amount. This is a useful property in the context of neural nets, because it means the network's computations behave in a well-defined way as we transform the inputs.

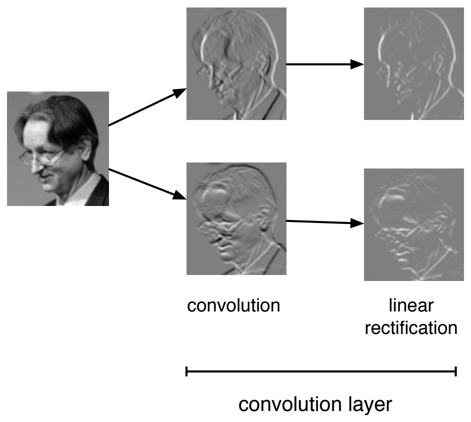


Figure 5: Detecting horizontal and vertical edge features.

2.3 Convolutional feature detection

As alluded to above, convolutions are even more powerful when they're paired with nonlinearities. A sequence of convolutions can only compute a linear function, but a sequence of convolutions alternated with nonlinearities can do fancier things. E.g., consider the following sequence of operations:

1. Convolve the image with a horizontal edge filter
2. Apply the linear rectification nonlinearity

$$\phi(z) = \begin{cases} z & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases} \tag{8}$$

3. Blur the result.

This sequence of steps, shown in Figure 5, gives a map of horizontalness in various parts of an image; the same can be done for verticalness. You can hopefully imagine this being a useful feature for further processing. Because the resulting output can be thought of as a map of the feature strength over parts of an image, we refer to it as a **feature map**.

3 Convolution layers

We just saw that a convolution, followed by a nonlinear activation function, followed by another convolution, could compute something interesting. This motivates the **convolution layer**, a neural net layer which computes convolutions followed by a nonlinear activation function. Since convolution layers can be thought of as doing feature detection, they're sometimes referred to as **detection layers**. First, let's see how we can think about convolution in terms of units and connections.

Confusingly, the way they're standardly defined, convolution layers don't actually compute convolutions, but a closely related operation called **filtering**:

$$(x \star w)[t] = \sum_{\tau} x[t + \tau]\, w[\tau]. \tag{9}$$

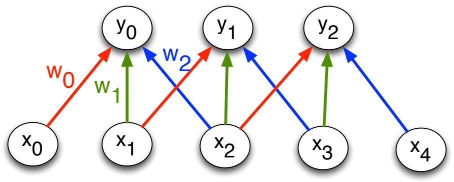


Figure 6: A convolution layer, shown in terms of units and connections.

Like the name suggests, filtering is essentially like flip-and-filter, but without the flipping. (I.e., $x * w = x \star \mathrm{flip}(w)$.) The two operations are basically equivalent — the difference is just a matter of how the filter (or kernel) is represented.

In the above example, we computed a single feature map, but just as we normally use more than one hidden unit in fully connected layers, convolution layers normally compute multiple feature maps $z_1, \ldots, z_M$. The input layers also consist of multiple feature maps $x_1, \ldots, x_D$; these could be different color channels of an RGB image, or feature maps computed by another convolution layer. There is a separate filter w_{ij} associated with each pair of an input and output feature map. The activations are computed as follows:

$$z_i = \sum_j x_j \star w_{ij} \tag{10}$$

$$h_i = \phi(z_i) \tag{11}$$

The activation function ϕ is applied elementwise.

We can think about filtering as a layer of a neural network by thinking of the elements of x and $x * w$ as units, and the elements of w as connection weights. Such an interpretation is visualized in Figure 6 for a one-dimensional example. Each of the units in this network computes its activations in the standard way, i.e. by summing up each of the incoming units multiplied by their connection weights. This shows that a convolution layer is like a fully connected layer, except with two additional features:

- **Sparse connectivity**: not every input unit is connected to every output unit.
- **Weight sharing**: the network's weights are each shared between multiple connections.

Missing connections can be thought of as connections with weight 0. This highlights an important fact: *any function computed by a convolution layer can be computed by a fully connected layer.*

This means convolution layers don't increase the representational capacity, relative to a fully connected layer with the same number of input and output units. But they can reduce the numbers of weights and connections. For instance, suppose we have 32 input feature maps and 16 output feature maps, all of size 50×50, and the filters are of size 5×5. (These are all plausible sizes for a conv net.) The number of weights for the convolution layer is

$$5 \times 5 \times 16 \times 32 = 12,800.$$

The number of connections is approximately

$$50 \times 50 \times 5 \times 5 \times 16 \times 32 = 32 \text{ million}.$$

By contrast, the number of connections (and hence also the number of weights) required for a fully connected layer with the same set of units would be

$$(32 \times 50 \times 50) \times (16 \times 50 \times 50) = 3.2 \text{ billion}.$$

Hence, using the convolutional structure reduces the number of connections by a factor of 100 and the number of weights by almost a factor of a million!

4 Pooling layers

In the introduction to this lecture, we observed that a neural network's classifications ought to be invariant to small transformations of an image, such as shifting it by a few pixels. In order to achieve invariance, we introduce another kind of layer: the **pooling layer**. Pooling layers summarize (or compress) the feature maps of the previous layer by computing a simple function over small regions of the image. Most commonly, this function is taken to be the maximum, so the operation is known as **max-pooling**.

Suppose we have input feature maps $x_1, \ldots, x_N$. Each unit of the output map computes the maximum over some region (called a **pooling group**) of the input map. (Typically, the region could be 3×3.) In order to shrink the representation, we don't consider all offsets, but instead we space them by a **stride** S along each dimension. This results in the representation being shrunk by a factor of approximately S along each dimension. (A typical value for the stride is 2.)

Figure 7 shows an example of how pooling can provide partial invariance to translations of the input.

Pooling also has the effect of increasing the size of units' **receptive fields**, or the regions of the input image which influence their activations. For instance, consider the network architecture in Figure 8, which alternates between convolution and pooling layers. Suppose all the filters are 5×5 and the pooling layer uses a stride of 2. Then each unit in the first convolution layer has a receptive field of size 5×5. But each unit in the second convolution layer has a receptive field of size approximately 10×10, since it does 5×5 filtering over a representation which was shrunken by a factor of 2 along each dimension. A third convolution layer would have 20×20 receptive fields. Hence, pooling allows small filters to account for information over large regions of an image.

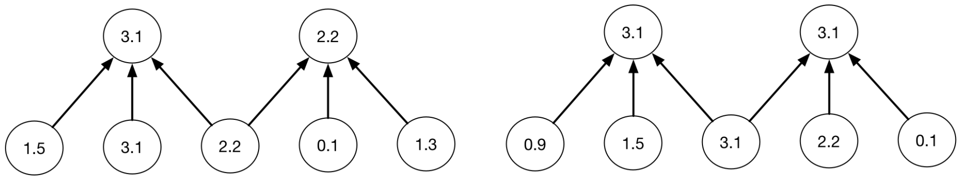


Figure 7: An example of how pooling can provide partial invariance to translations of the input. Observe that the first output does not change, since the maximum value remains within its pooling group.

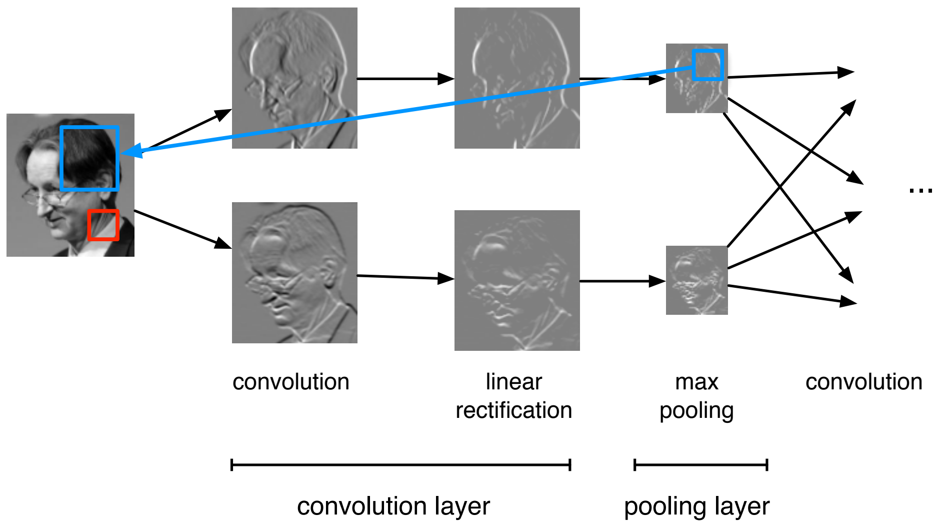


Figure 8: Schematic of a conv net with convolution and pooling layers. Pooling layers expand the receptive fields of units in subsequent convolution layers.