Convolutional Neural Networks

So far, we have studied what are called *fully connected* neural networks, in which all of the units at one layer are connected to all of the units in the next layer. This is a good arrangement when we don't know anything about what kind of mapping from inputs to outputs we will be asking the network to learn to approximate. But if we *do* know something about our problem, it is better to build it into the structure of our neural network. Doing so can save computation time and significantly diminish the amount of training data required to arrive at a solution that generalizes robustly.

One very important application domain of neural networks, where the methods have achieved an enormous amount of success in recent years, is signal processing. Signals might be spatial (in two-dimensional camera images or three-dimensional depth or CAT scans) or temporal (speech or music). If we know that we are addressing a signal-processing problem, we can take advantage of *invariant* properties of that problem. In this chapter, we will focus on two-dimensional spatial problems (images) but use one-dimensional ones as a simple example. Later, we will address temporal problems.

Imagine that you are given the problem of designing and training a neural network that takes an image as input, and outputs a classification, which is positive if the image contains a cat and negative if it does not. An image is described as a two-dimensional array of *pixels*, each of which may be represented by three integer values, encoding intensity levels in red, green, and blue color channels.

> A *pixel* is a "picture element."

There are two important pieces of prior structural knowledge we can bring to bear on this problem:

- **Spatial locality:** The set of pixels we will have to take into consideration to find a cat will be near one another in the image.

> So, for example, we won't have to consider some combination of pixels in the four corners of the image, in order to see if they encode cat-ness.

- **Translation invariance:** The pattern of pixels that characterizes a cat is the same no matter where in the image the cat occurs.

> Cats don't look different if they're on the left or the right side of the image.

We will design neural network structures that take advantage of these properties.

1 Filters

We begin by discussing *image filters*. An image filter is a function that takes in a local spatial neighborhood of pixel values and detects the presence of some pattern in that data.

Let's consider a very simple case to start, in which we have a 1-dimensional binary "image" and a filter F of size two. The filter is a vector of two numbers, which we will move along the image, taking the dot product between the filter values and the image values at each step, and aggregating the outputs to produce a new image.

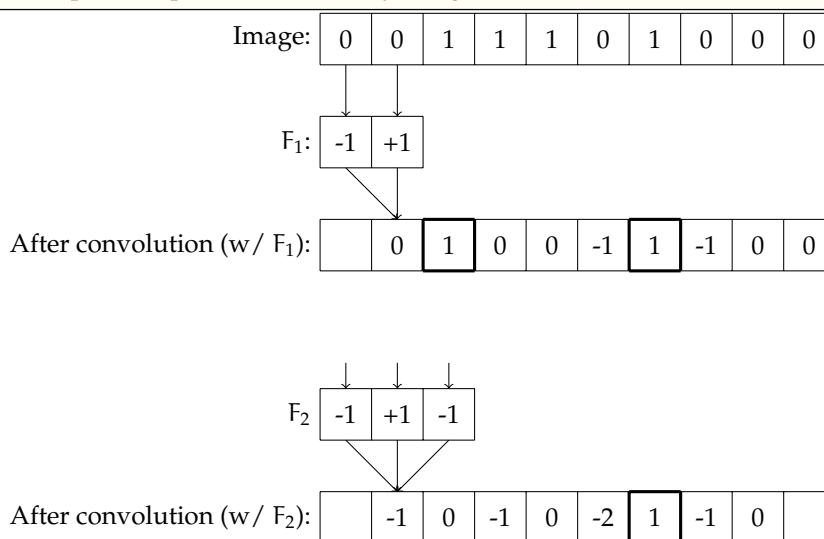Let X be the original image, of size d; then pixel i of the the output image is specified by

$$Y_i = F \cdot (X_{i-1}, X_i) \ .$$

To ensure that the output image is also of dimension d, we will generally "pad" the input image with 0 values if we need to access pixels that are beyond the bounds of the input image. This process of applying the filter to the image to create a new image is called "convolution."

If you are already familiar with what a convolution is, you might notice that this definition corresponds to what is often called a correlation and not to a convolution. Indeed, correlation and convolution refer to different operations in signal processing. However, in the neural networks literature, most libraries implement the correlation (as described in this chapter) but call it convolution. The distinction is not significant; in principle, if convolution is required to solve the problem, the network could learn the necessary weights. For a discussion of the difference between convolution and correlation and the conventions used in the literature you can read section 9.1 in this excellent book: `https://www.deeplearningbook.org`.

Here is a concrete example. Let the filter $F_1 = (-1, +1)$. Then given the first image below, we can convolve it with filter F_1 to obtain the second image. You can think of this filter as a detector for "left edges" in the original image—to see this, look at the places where there is a 1 in the output image, and see what pattern exists at that position in the input image. Another interesting filter is $F_2 = (-1, +1, -1)$. The third image below shows the result of convolving the first image with F_2.

> **Study Question:** Convince yourself that filter F_2 can be understood as a detector for isolated positive pixels in the binary image.

Image:	0	0	1	1	1	0	1	0	0	0

F_1:	-1	+1

After convolution (w/ F_1):		0	1	0	0	-1	1	-1	0	0

F_2	-1	+1	-1

| After convolution (w/ F_2):		-1	0	-1	0	-2	1	-1	0	

Two-dimensional versions of filters like these are thought to be found in the visual cortex of all mammalian brains. Similar patterns arise from statistical analysis of natural

Unfortunately in AI/ML/CS/Math, the word "filter" gets used in many ways: in addition to the one we describe here, it can describe a temporal process (in fact, our moving averages are a kind of filter) and even a somewhat esoteric algebraic structure.

And filters are also sometimes called *convolutional kernels*.

images. Computer vision people used to spend a lot of time hand-designing *filter banks*. A filter bank is a set of sets of filters, arranged as shown in the diagram below.
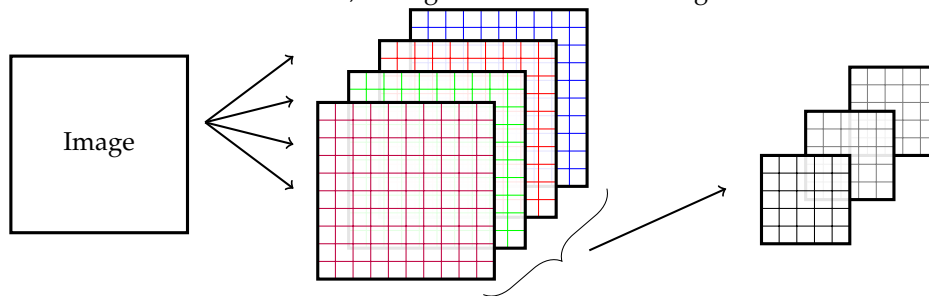

All of the filters in the first group are applied to the original image; if there are k such filters, then the result is k new images, which are called *channels*. Now imagine stacking all these new images up so that we have a cube of data, indexed by the original row and column indices of the image, as well as by the channel. The next set of filters in the filter bank will generally be *three-dimensional*: each one will be applied to a sub-range of the row and column indices of the image and to all of the channels.

These 3D chunks of data are called *tensors*. The algebra of tensors is fun, and a lot like matrix algebra, but we won't go into it in any detail.

Here is a more complex example of two-dimensional filtering. We have two 3×3 filters in the first layer, f_1 and f_2. You can think of each one as "looking" for three pixels in a row, f_1 vertically and f_2 horizontally. Assuming our input image is $n \times n$, then the result of filtering with these two filters is an $n \times n \times 2$ tensor. Now we apply a tensor filter (hard to draw!) that "looks for" a combination of two horizontal and two vertical bars (now represented by individual pixels in the two channels), resulting in a single final $n \times n$ image.

> There is a popular piece of neural-network software called *Tensorflow* that makes operations on tensors easy.

> When we have a color image as input, we treat it as having 3 channels, and hence as an $n \times n \times 3$ tensor.
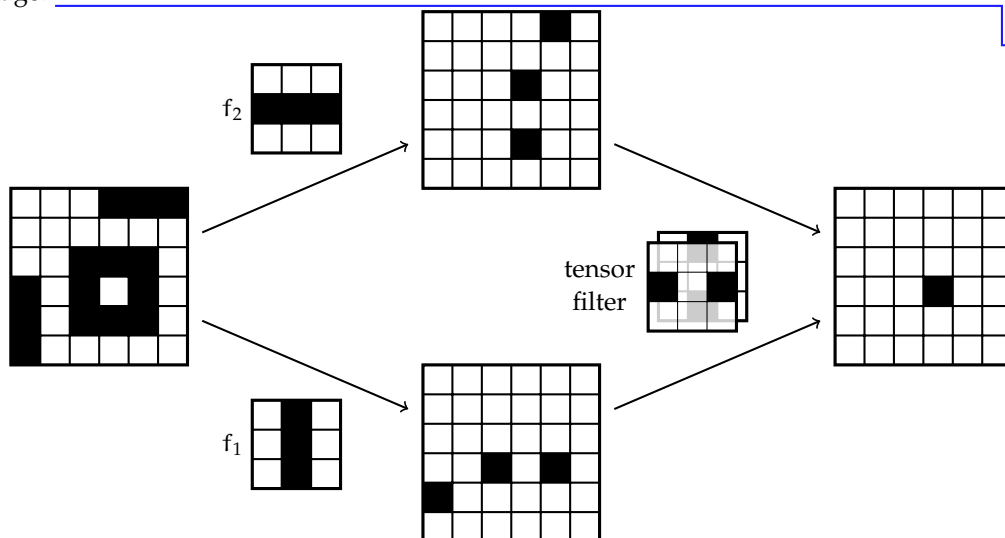
We are going to design neural networks that have this structure. Each "bank" of the filter bank will correspond to a neural-network layer. The numbers in the individual filters will be the "weights" (plus a single additive bias or offset value for each filter) of the network, which we will train using gradient descent. What makes this interesting and powerful (and somewhat confusing at first) is that the same weights are used many many times in the computation of each layer. This *weight sharing* means that we can express a transformation on a large image with relatively few parameters; it also means we'll have to take care in figuring out exactly how to train it!

We will define a filter layer l formally with:

- *number* of filters m^l;

- *size* of one filter is $k^l \times k^l \times m^{l-1}$ plus 1 bias value (for this one filter);

- *stride* s^l is the spacing at which we apply the filter to the image; in all of our examples so far, we have used a stride of 1, but if we were to "skip" and apply the filter only at odd-numbered indices of the image, then it would have a stride of two (and produce a resulting image of half the size);

- *input tensor size* $n^{l-1} \times n^{l-1} \times m^{l-1}$

- *padding*: p^l is how many extra pixels – typically with value 0 – we add around the edges of the input. For an input of size $n^{l-1} \times n^{l-1} \times m^{l-1}$, our new effective input size with padding becomes $(n^{l-1} + 2 * p^l) \times (n^{l-1} + 2 * p^l) \times m^{l-1}$.

> For simplicity, we are assuming that all images and filters are square (having the same number of rows and columns). That is in no way necessary, but is usually fine and definitely simplifies our notation.

This layer will produce an output tensor of size $n^l \times n^l \times m^l$, where $n^l = \lceil (n^{l-1} + 2 * p^l - (k^l - 1))/s^l \rceil$.[1] The weights are the values defining the filter: there will be m^l different $k^l \times k^l \times m^{l-1}$ tensors of weight values; plus each filter may have a bias term, which means there is one more weight value per filter. A filter with a bias operates just like the filter examples above, except we add the bias to the output. For instance, if we incorporated a bias term of 0.5 into the filter F_2 above, the output would be $(-0.5, 0.5, -0.5, 0.5, -1.5, 1.5, -0.5, 0.5)$ instead of $(-1, 0, -1, 0, -2, 1, -1, 0)$.

This may seem complicated, but we get a rich class of mappings that exploit image structure and have many fewer weights than a fully connected layer would.

> **Study Question:** How many weights are in a convolutional layer specified as above?

> **Study Question:** If we used a fully-connected layer with the same size inputs and outputs, how many weights would it have?

2　Max Pooling

It is typical to structure filter banks into a *pyramid*, in which the image sizes get smaller in successive layers of processing. The idea is that we find local patterns, like bits of edges in the early layers, and then look for patterns in those patterns, etc. This means that, effectively, we are looking for patterns in larger pieces of the image as we apply successive filters. Having a stride greater than one makes the images smaller, but does not necessarily aggregate information over that spatial range.

> Both in engineering and in nature

Another common layer type, which accomplishes this aggregation, is *max pooling*. A max pooling layer operates like a filter, but has no weights. *You can think of it as a pure functional layer, like a ReLU layer in a fully connected network.* It has a filter size, as in a filter layer, but simply returns the maximum value in its field. Usually, we apply max pooling with the following traits:

> We sometimes use the term *receptive field* or just *field* to mean the area of an input image that a filter is being applied to.

- stride > 1, so that the resulting image is smaller than the input image; and

- $k \geqslant$ stride, so that the whole image is covered.

---

[1]Recall that $\lceil \cdot \rceil$ is the *ceiling* function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.

As a result of applying a max pooling layer, we don't keep track of the precise location of a pattern. This helps our filters to learn to recognize patterns independent of their location.

Consider a max pooling layer of stride = $k = 2$. This would map a $64 \times 64 \times 3$ image to a $32 \times 32 \times 3$ image. Note that max pooling layers do not have additional bias or offset values.

> **Study Question:** Maximilian Poole thinks it would be a good idea to add two max pooling layers of size k, one right after the other, to their network. What single layer would be equivalent?

3 Typical architecture
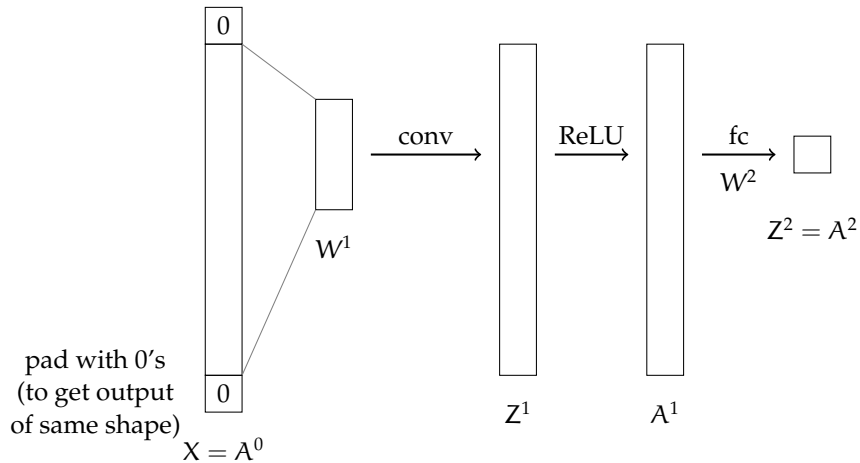
Here is the form of a typical convolutional network:

Source: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html
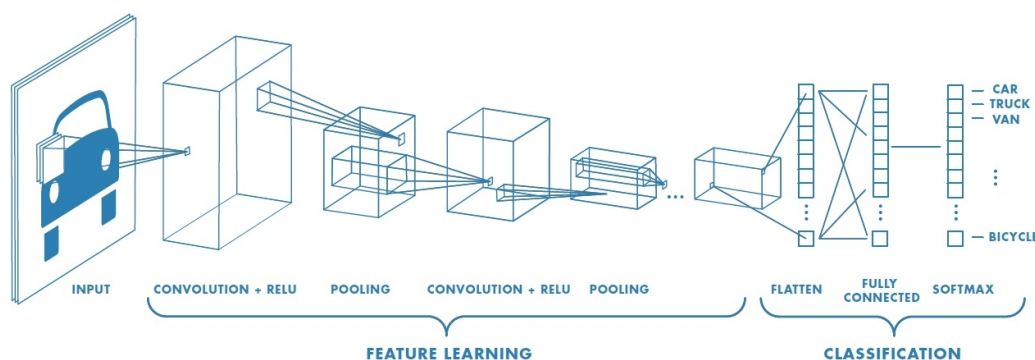
After each filter layer there is generally a ReLU layer; there maybe be multiple filter/ReLU layers, then a max pooling layer, then some more filter/ReLU layers, then max pooling. Once the output is down to a relatively small size, there is typically a last fully-connected layer, leading into an activation function such as softmax that produces the final output. The exact design of these structures is an art—there is not currently any clear theoretical (or even systematic empirical) understanding of how these various design choices affect overall performance of the network.

The critical point for us is that this is all just a big neural network, which takes an input and computes an output. The mapping is a differentiable function of the weights, which means we can adjust the weights to decrease the loss by performing gradient descent, and we can compute the relevant gradients using back-propagation!

> Well, the derivative is not continuous, both because of the ReLU and the max pooling operations, but we ignore that fact.

Let's work through a *very* simple example of how back-propagation can work on a convolutional network. The architecture is shown below. Assume we have a one-dimensional single-channel image, of size $n \times 1 \times 1$ and a single $k \times 1 \times 1$ filter (where we omit the filter bias) in the first convolutional layer, denoted "conv" in the figure below. Then we pass it through a ReLU layer, and finally through a fully-connected layer, denoted "fc" below, with no additional activation function on the output.

For simplicity assume k is odd, let the input image $X = A^0$, and assume we are using squared loss. Then we can describe the forward pass as follows:

$$Z_i^1 = W^{1^T} \cdot A^0_{[i-\lfloor k/2 \rfloor : i + \lfloor k/2 \rfloor]}$$

$$A^1 = \text{ReLU}(Z^1)$$

$$A^2 = Z^2 = W^{2^T} A^1$$

$$L(A^2, y) = (A^2 - y)^2$$

Study Question: For a filter of size k, how much padding do we need to add to the top and bottom of the image?

How do we update the weights in filter W^1?

$$\frac{\partial \text{loss}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial \text{loss}}{\partial A^1}$$

- $\partial Z^1 / \partial W^1$ is the $k \times n$ matrix such that $\partial Z_i^1 / \partial W_j^1 = X_{i-\lfloor k/2 \rfloor + j - 1}$. So, for example, if $i = 10$, which corresponds to column 10 in this matrix, which illustrates the dependence of pixel 10 of the output image on the weights, and if $k = 5$, then the elements in column 10 will be $X_8, X_9, X_{10}, X_{11}, X_{12}$.

- $\partial A^1 / \partial Z^1$ is the $n \times n$ diagonal matrix such that

$$\partial A_i^1 / \partial Z_i^1 = \begin{cases} 1 & \text{if } Z_i^1 > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $\partial \text{loss} / \partial A^1 = \partial \text{loss} / \partial A^2 \cdot \partial A^2 / \partial A^1 = 2(A^2 - y)W^2$, an $n \times 1$ vector

Multiplying these components yields the desired gradient, of shape $k \times 1$.