Deep Convolutional Neural Nets

Carlo Tomasi

November 29, 2023

Neural nets are a class of predictors that have been shown empirically to achieve very good performance on tasks whose inputs are images, speech, or audio signals. They have also been applied to inputs of other types, with varied results. The reasons why these predictors work so well are still unclear. What *is* clear is that they are very expressive, in the sense that the hypothesis space that they define is very large. Theorems show that *any* Lipschitz function[1] from a hypercube in $\mathbb{R}^d$ to an interval in $\mathbb{R}$ can be approximated arbitrarily closely (that is, within any pre-specified $\epsilon > 0$) with a neural net.

While expressive power is good, we know that it entails dangers. First, the approximation theorems just mentioned are practically irrelevant, because the computational complexity of a neural net needed to achieve a given accuracy ϵ turns out to grow exponentially with the dimension d of the input space. Thus, neural nets can approximate anything, but at an unrealistically high cost. Second, the fact that an approximator *exists* in a given hypothesis class does not mean that we know how to find it: Training a predictor amounts to minimizing the empirical risk L_T over the training set T, and if L_T is not a convex function, finding a global minimum is generally computationally intractable. Third, and perhaps most importantly for machine learning, a high expressive power leads to overfitting, as we know well by now.

The last consideration implies that training a neural net requires large training sets, that is, that the sample complexity of neural nets is high. Indeed, perhaps the greatest hurdle to the widespread use of neural nets is the cost of collecting, and most importantly annotating, data sets with millions of samples. Arguably, the most important reason for the success of neural nets is not so much the nets themselves, but the emergence of the Amazon Mechanical Turk, a crowdsourcing marketplace made available by Amazon. With the Turk, you can post millions of data samples on a web site, and let any user anywhere annotate the data for you, for a reward of a few cents per sample. There would be no successful deep neural nets without the Turk. More recently, several companies have emerged that offer annotation as a paid service.

At the same time, even the large sizes of current training nets cannot fully explain their ability to generalize: The inputs to neural nets often have dimensionality d in the tens or hundreds of thousands, and no amount of data under the sun can keep up with the exponential growth of X with d. There must be deeper reasons at play, that have to do with (i) the special structure of image space (or audio space, and so forth); (ii) the specialized architectures proposed for neural nets; and (iii) tricks and techniques used to regularize training.

In summary, neural nets are very expressive and data hungry. In spite of their expressiveness, they often generalize better than one would predict. We don't fully know why, although theoreti-

[1]Somewhat loosely speaking, a differentiable function is Lipschitz when its gradient is uniformly bounded by a constant. This notion can be defined more generally without reference to differentiability.

cians are making constant progress towards an answer. Because of the still immature degree of theoretical understanding of neural nets, the treatment in these notes will have to be based on half-baked intuitions and empirical evidence.

This note covers one particular way to build a particular type of deep feed-forward network. Such a network can be used for either classification or regression, and we will focus on classification. More variants and details can be found in many books or articles on neural nets [1], convolutional neural nets [6], and deep learning in general [2, 3].

A later note on training will describe how to determine the parameters (weights) of a deep feed-forward network of a given structure and for a given classification task.

1 Circuits

Suppose that you want to implement a predictor $h : X \to Y$ on a computer. There are various ways to describe the implementation of h. The one we are most familiar with is in terms of an *algorithm*, a sequence of steps to be performed in sequence over time. Another way is to specify a *circuit*, a computational model that mimics how electrical circuits are built. A (computational) circuit is made of a possibly large number of *gates*, each of which implements one of a small set of predefined functions. For example, logical circuits are made mostly out of NAND (not-and) gates, which when combined can produce any Boolean function.

Circuits and algorithms are equivalent to each other: You can build a Boolean function by buying NAND gates at Radio Shack and wiring them together, or you can write a piece of Python code that simulates the circuit. Since the simulation simulates one gate at a time, it may take a long time to simulate a complex circuit. You can also go the other way: Given an algorithm, come up with a circuit that implements the same (Boolean) function. This must be possible: After all, a computer is a large circuit that runs algorithms. You may object that computers compute more than just Boolean functions, including, say, real-valued functions, but they really do not: A number is represented by a finite string of bits in a computer, so the output is still a set of Boolean variables, which are functions of the Boolean variables that represent the inputs.

A neural network is a class of algorithms that are typically described as circuits, and are made by *neurons*. A set of neurons is said to form a *layer* if each neuron in the set receives the same inputs. A *neural network* is a cascade of layers, in which the outputs from one layer are the inputs to the next. The network is *deep* if it has many layers. Every neuron has *parameters*, so a neural network has many parameters. The network is *convolutional* if the parameters of the neurons in each layer are constrained in a special way. The Sections that follow define these concepts. *Training* a neural network amounts to finding the parameters that minimize the training risk. Training is discussed in a later note.

2 Neurons

A *neuron* (in the computational sense) is a function $\mathbb{R}^d \to \mathbb{R}$ of the form

$$y = \rho(a(\mathbf{x})) \quad \text{where} \quad a = \mathbf{w}^T \tilde{\mathbf{x}} \quad , \quad \tilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix} .$$

The entries of the vector $\mathbf{w} \in \mathbb{R}^{d+1}$ are called the *weights*, and the *activation function* is a nonlinear and weakly monotonic function $\mathbb{R} \to \mathbb{R}$. The input $a(\mathbf{x})$ to ρ is called the *activation* of the neuron,

and the particular type of activation function

$$\rho(a) = \max(0, a)$$

is called the *Rectified Linear Unit* (ReLU, Figure 1).

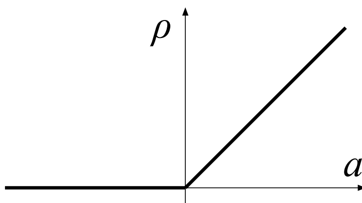

Figure 1: The Rectified Linear Unit (ReLU).

We view the tilde (as in $\tilde{\mathbf{x}}$) as an operator: Given any vector $\mathbf{x}$, this operator appends a 1 at the end of $\mathbf{x}$.

The activation can be rewritten as follows

$$a = \mathbf{v}^T\mathbf{x} + b \quad \text{where} \quad \mathbf{v}^T = [w_1, \ldots, w_d] \quad \text{and} \quad b = w_{d+1} \; ,$$

and is an inner product between a *gain*[2] vector $\mathbf{v}$ and the input $\mathbf{x}$ plus a *bias* b. Figure 2 shows a neuron in diagrammatic form.

For different inputs $\mathbf{x}$ of the same magnitude[3], the activation is maximum when $\mathbf{x}$ is parallel to $\mathbf{v}$, and the latter can be viewed as a *pattern* or *template* to which $\mathbf{x}$ is compared. The bias b then raises or lowers the activation before it is passed through the activation function.

The ReLU will *respond* (that is, return a nonzero output) if the inner product $\mathbf{v}^T\mathbf{x}$ is greater than $-b$ (so that a is positive), and the response thereafter increases with the value of a. So the negative of the bias can be viewed as a *threshold* that the inner product between pattern and input must exceed before it is deemed to be significant, and the neuron can be viewed as a *score function* that measures the similarity of the suitably normalized input $\mathbf{x}$ to the pattern $\mathbf{v}$ when the similarity is significant (that is, greater than $-b$). When the similarity is not significant, the neuron does not respond.

A *pattern classifier* would add a stage that decides if the score is large enough to declare the input $\mathbf{x}$ to contain the pattern represented by $\mathbf{v}$. So another way to view a neuron is a pattern classifier without the decision stage.

3 Two-Layer Neural Nets

A neural-net *layer* is a vector of $d^{(1)}$ neurons, that is, a function $\mathbb{R}^d \to \mathbb{R}^{d^{(1)}}$

$$\mathbf{y} = \rho(\mathbf{a}(\mathbf{x})) \quad \text{where} \quad \mathbf{a}(\mathbf{x}) = W\tilde{\mathbf{x}} \; ,$$

the weight matrix W is $d^{(1)} \times (d+1)$, and the activation function ρ is applied to each entry of the activation vector $\mathbf{a}(\mathbf{x}) \in \mathbb{R}^{d^{(1)}}$. So a neural-net layer can be viewed as a *bank* of pattern scoring devices, one pattern per neuron. Figure 3 illustrates.

[2] Gains are often called weights as well.

[3] As measured by its Euclidean norm $\|\mathbf{x}\|$.

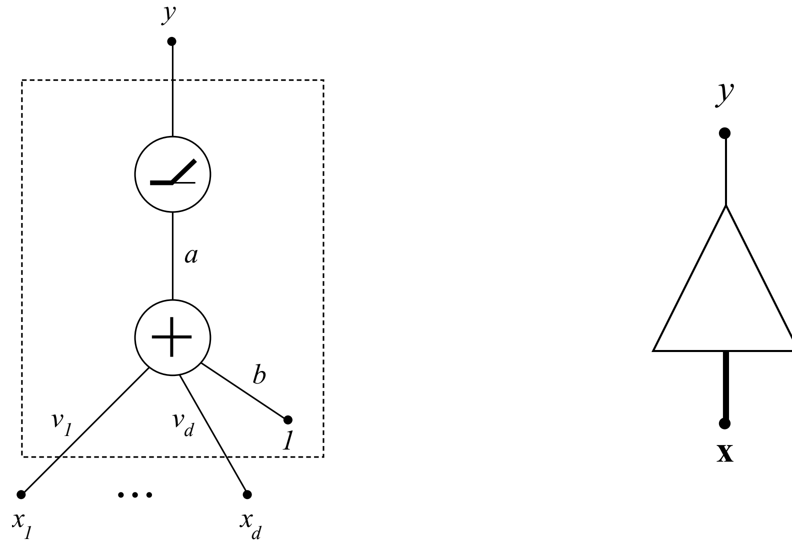

Figure 2: The internal structure of a neuron (left) and a neuron as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.

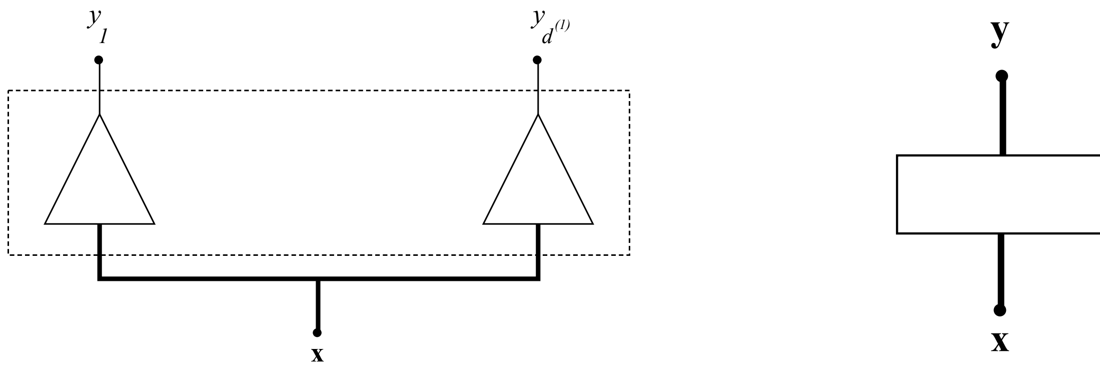

Figure 3: The internal structure of a layer (left) and a layer as a black box (right). The black box corresponds to the part inside the dashed rectangle on the left.

To compute the output of a layer from its input one needs to perform $d^{(1)}d$ multiplications and as many additions to compute the activation vector, and then compute the activation function $d^{(1)}$ times. So if $d^{(1)}$ is of the same order of magnitude as d, the cost of this computation is quadratic in the size d of the input $\mathbf{x}$. Even more importantly, there are $O(d^2)$ parameters (the entries of W) that need to be determined when the layer is trained.

A *two-layer neural net* is a cascade of two layers,

$$\begin{aligned} \mathbf{x}^{(1)} &= \rho(W^{(1)}\tilde{\mathbf{x}}) \\ \mathbf{y} &= \rho_y(W^{(2)}\tilde{\mathbf{x}}^{(1)}) \end{aligned}$$

where the activation function ρ_y can be different from ρ and $W^{(2)}$ is $d^{(2)} \times (d^{(1)} + 1)$.

It can be proven [7] that any mapping $\mathbb{R}^d \to \mathbb{R}^{d^{(2)}}$ that is Lebesgue-integrable and has Lebesgue-integrable Fourier transform[4] can be approximated to any finite degree of accuracy over a hypercube in $\mathbb{R}^d$ with a two-layer neural net where ρ_y is the identity function and ρ is the ReLU. This result, along with similar ones for other activation functions [4], shows that two-layer neural nets are *universal approximators*.

However, a two-layer approximator to a given function f may be very expensive to implement, and may have a number of parameters that is exponential in the desired accuracy. This makes both computational complexity and sample complexity unaffordable. Deep neural nets are introduced in the hope that they lead to more efficient approximations for the types of function of interest, as discussed in the next two sections.

4 Convolutional Layers

A neuron matches the input $\mathbf{x}$ to a pattern $\mathbf{v}$. What should the patterns in an image recognition system be? One could make $\mathbf{x}$ be the entire image, with its pixels strung into a vector, and then $\mathbf{v}$ could be an image (in vector form as well) of the object to be recognized—say, your grandmother's face. This net would not work well, as your grandmother's face could show up in images that look very different from $\mathbf{v}$ because of viewpoint, lighting, facial expression, other objects or people in the image, and other causes of discrepancy.

Instead, observe that faces typically have eyes, noses, ears, hair, and wrinkles—especially for an older person. These features can be analyzed in turn in terms of image edges, corners, curved segments, small dark regions, and so forth. This suggest building a *hierarchy* of patterns, where higher-level ones are made of lower level ones, and only the lowest-level patterns are made directly out of pixels from the input image. At each level, each pattern should then take only a relatively small and compact part of the input in consideration: The input of each neuron should be relatively *local*.

In addition, many of the lower-level features appear multiple times in images and across objects, and this suggests that the same neuron could compute scores of patterns of its own type no matter where they appear in an image: the same detector could be *reused* over its domain.

Finally, if higher-level patterns are somehow made somewhat insensitive to exactly where in the image the relevant lower-level patterns occur, then the overall system would be able to recognize your grandmother's face even in the presence of at least some amount of spatial variation. A

[4] Just think of these as mild requirements on the mapping. It is not important for our purposes to know what they mean.

hierarchy with many levels may be able to achieve this even more easily, since a small amount of *resilience to spatial variation* in each layer might result in more significant resilience once it is compounded across layers.

These notions of locality, reuse, and resilience to spatial variations suggest imposing a special structure on a neural-net layer that works on images or signals. Before looking at the overall structure, we return to the points of "locality" and "reuse" and introduce the notion of *correlation*. Correlation is closely related to another concept called "convolution," which has given *Convolutional Neural Nets (CNNs)* their name.

4.1 One-Dimensional Correlation

Consider the affine part of a single layer

$$\mathbf{a} = W\tilde{\mathbf{x}} = V\mathbf{x} + \mathbf{b}$$

where $\mathbf{x} \in \mathbb{R}^d$ and $\mathbf{a} \in \mathbb{R}^e$, so that the gain matrix V is $e \times d$, and $\mathbf{b} \in \mathbb{R}^e$ is a vector of biases. We saw that if both the input $\mathbf{x}$ and the pattern $\mathbf{v}_i$ were normalized to have unit norm, then entry number i of the product $V\mathbf{x}$,

$$s_i = \mathbf{v}_i^T \mathbf{x}$$

for $i = 0, \ldots, e-1$ could be viewed as a score that measures how similar vector $\mathbf{x}$ is to pattern $\mathbf{v}_i$.

> **Warnings:** In what follows, we reason about pattern $\mathbf{v}_i$ as if we had to design its values, in order to understand the issues involved. In reality, the values in $\mathbf{v}_i$ will be determined by the neural-network training algorithm so as to minimize the training risk.
>
> The example below, inspired by audio signal analysis, is unrealistic in many ways. It is simply used to make a mathematical point, not to examine audio-signal analysis.

Suppose that we analyze a clip $\mathbf{x}$ of $d = 25$ sound samples, and we want to determine whether the clip represents the attack of a drumbeat. One way to figure that out is to record a drumbeat attack $\mathbf{g}$, which might look like the sequence of 25 samples shown in Figure 4 (a), normalize its values, and compare it to the input sequence $\mathbf{x}$, also normalized, by an inner product.

As an example, the two clips shown in Figure 4 (b, c) yield inner products of about 0.999 and 0.241, consistently with the fact that the sequences in (a, b) are much more similar to each other than those in (a, c). We could then compare this inner product with a threshold $-b$, and if

$$\mathbf{g}^T \mathbf{x} \geq -b \; ,$$

that is, if

$$a = \mathbf{g}^T \mathbf{x} + b \geq 0 \; ,$$

we could send the value a, which represents the amount by which the inner product exceeds the threshold, to other modules for further processing. If a is negative, we send 0, meaning that we decided that the input $\mathbf{x}$ is uninteresting. In other words, if $\mathbf{g}$ and $\mathbf{x}$ are the normalized versions of these two sequences, this comparison unit would be a neuron, with output

$$y = \rho(a) = \max(0, \mathbf{g}^T \mathbf{x} + b) \; .$$

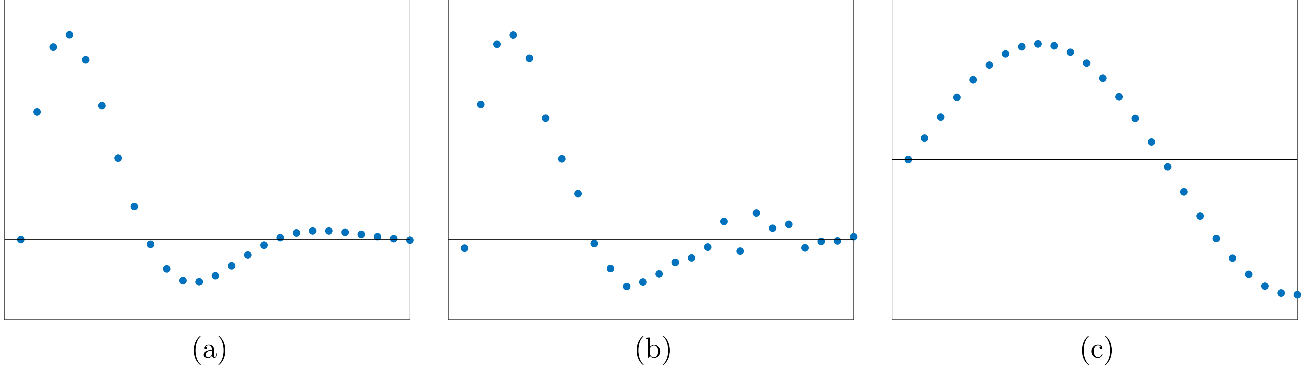

Figure 4: (a) A sequence of samples that represent the attack of a note in a sound clip. (b, c) Two other sequences of sound samples.

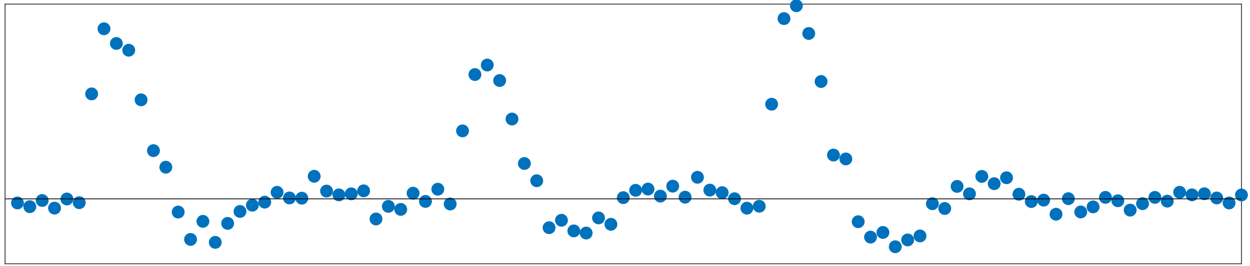

Figure 5: A longer clip of sound samples.

If we had, say, four different types of instruments (drum, guitar, bass, piano), we could have four different prototype samples $\mathbf{g}_0, \mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$, one per instrument, and have four separate neurons that "recognize" each instrument. These neurons would form a layer, with output

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \rho(V\mathbf{x} + \mathbf{b}) \quad \text{where} \quad V = \begin{bmatrix} \mathbf{g}_0^T \\ \mathbf{g}_1^T \\ \mathbf{g}_2^T \\ \mathbf{g}_3^T \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} .$$

Suppose now that we have a longer clip $\mathbf{x}$ of sound samples, such as the sequence in Figure 5. Going back to a single instrument (drum), the problem now is not to determine whether the entire clip is a drumbeat, but rather to *find* all the drumbeats in the clip.

If the clip $\mathbf{x}$ is, say, $d = 100$ samples long, since a drumbeat attack $\mathbf{g}$ is $k = 25$ samples long, we could have

$$e = d - k + 1 = 76$$

separate neurons, each specializing on a k-sample long subsequence of the clip by taking the inner product of $\mathbf{g}$ with that subsequence. The first neuron looks at samples 1 through 25 of $\mathbf{x}$, the second looks at samples 2 through 26, and so forth, and the 76-th neuron looks at samples 76 through 100 of $\mathbf{x}$.

If we think of all of $\mathbf{x}$ as the input to each of the 76 neurons, then the neurons form a layer. To "specialize" on samples i to $i+24$, neuron i has the following 100 gains:

$$\mathbf{v}_i^T = [\underbrace{0,\ldots,0}_{i-1},\underbrace{g_0,\ldots,g_{24}}_{\mathbf{g}},\underbrace{0,\ldots 0}_{76-i}]$$

and if we arrange these 76 row vectors into a matrix we obtain the 76×100 gain matrix

$$V = \begin{bmatrix} g_0 & \cdots & g_{24} & 0 & 0 & \cdots & 0 \\ 0 & g_0 & \cdots & g_{24} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & & \ddots & 0 \\ 0 & \cdots & \cdots & 0 & g_0 & \cdots & g_{24} \end{bmatrix} .$$

Each row of this matrix represents a *local* computation in that each neuron (row of V) only "sees" a small part of the input $\mathbf{x}$, corresponding to the nonzero entries in the row. This matrix also *reuses* the same set $\mathbf{g}$ of coefficients, which reoccur in every row.

Storing the entire matrix is wasteful, given all the zeros. A more compact computation of

$$\mathbf{z} = V\mathbf{x}$$

can be represented row by row as follows:

$$z_i = \sum_{a=0}^{24} g_a x_{i+a} \quad \text{for} \quad i = 0,\ldots,75 .$$

More generally, with k samples in $\mathbf{g}$,

$$z_i = \sum_{a=0}^{k-1} g_a x_{i+a} \quad \text{for} \quad i = 0,\ldots,e-1 = d-k . \tag{1}$$

This is the standard definition of inner product (or, in the language of matrices, row by column product), but focusing only on the nonzero terms, and with the acknowledgement that every row uses the same k coefficients g_a.

The operation defined in equation 1 is called the *(one-dimensional) correlation* of input $\mathbf{x}$ with *kernel* $\mathbf{g}$. Equation 1 reflects the order of computation that would be followed if the outputs z_i were computed in sequence: The kernel $\mathbf{g}$ is first ($i = 0$) aligned with the leftmost entries of $\mathbf{x}$. Corresponding entries in $\mathbf{g}$ and $\mathbf{x}$ are multiplied together and the products added up to yield z_0. The window is then slid by one position to the right, and the operation is repeated to compute z_1. The process ends when the right edge of $\mathbf{g}$ "hits" the right edge of $\mathbf{x}$.

Figure 6 illustrates the computation of one-dimensional correlation in the various forms introduced so far for the smaller case $d = 8$, $k = 3$ (so that $e = 8-3+1 = 6$):

$$z_i = \sum_{a=0}^{2} g_a x_{i+a} \quad \text{for} \quad i = 0,\ldots,5 .$$

If we now had four instruments, like earlier, we would stack the four corresponding matrices V_0 (drum), V_1 (guitar), V_2 (bass), v_3 (piano) in a third dimension. Each matrix has its own kernel $\mathbf{g}_c$ for $c = 0,1,2,3$. This third dimension corresponds to what are called *channels* in a convolutional neural network.

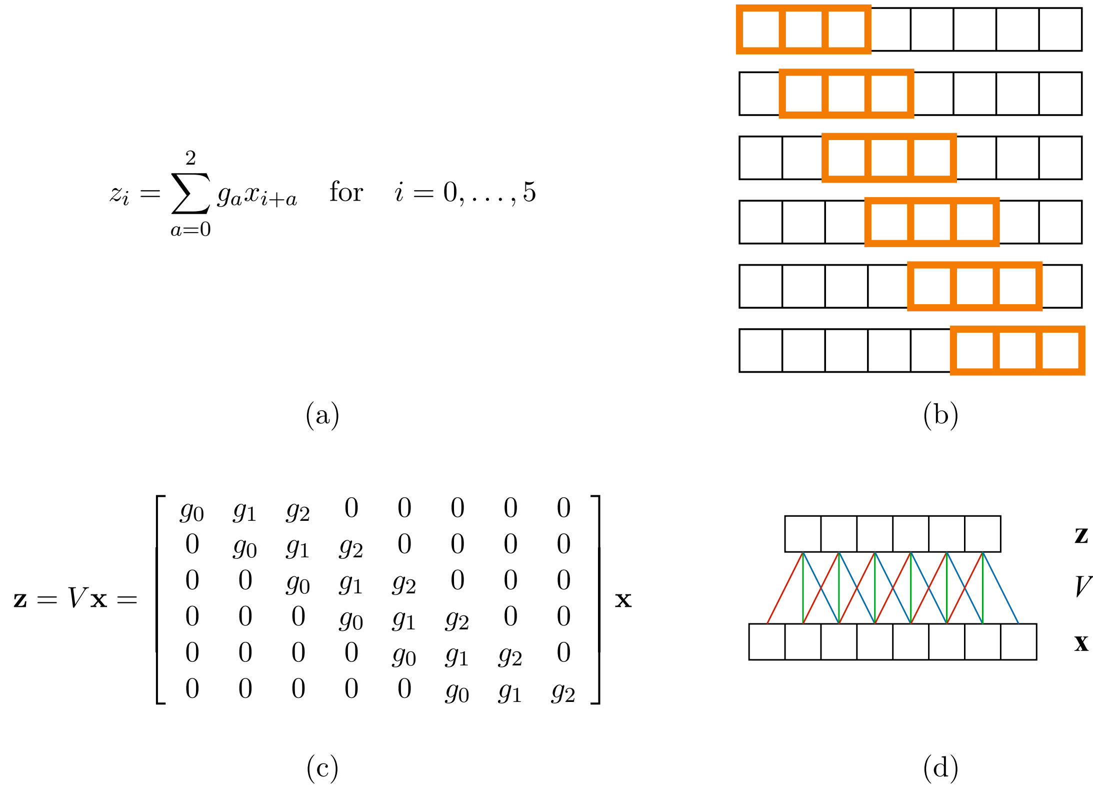

Figure 6: Four different views of a one-dimensional correlation from $\mathbf{x} \in \mathbb{R}^8$ to $\mathbf{z} \in \mathbb{R}^6$ with a kernel $\mathbf{g}$ of length $k = 3$. (a) The scalar view is a formula for the computation of each entry z_i of the output $\mathbf{z}$. (b) The sliding-window view: The kernel $\mathbf{g}$ (orange) is slid over all positions that overlap fully with the input $\mathbf{x}$ (black). At each position, the entries of $\mathbf{g}$ are multiplied with the corresponding entries of $\mathbf{x}$, and the products are added up to yield the output entry of $\mathbf{z}$. (c) Matrix representation. All rows contain the same entries, and the matrix is sparse. Three diagonals contain repetitions of one of g_0, g_1, g_2. (d) Circuit view: Red links correspond to multiplication by g_0, green by g_1, blue by g_2. The summation is left implicit.

4.2 Correlation and Convolution

Given a correlation kernel $\mathbf{g}$, let $\mathbf{r}$ be the kernel obtained by listing the entries of $\mathbf{g}$ in reverse order:

$$\mathbf{r} = [r_0, \ldots, r_{k-1}] = [g_{k-1}, \ldots, g_0] \; .$$

Then, the correlation of input $\mathbf{x}$ with kernel $\mathbf{g}$ is also called the *convolution* of $\mathbf{x}$ with $\mathbf{r}$ (and *vice versa*, of course).

There are many important reasons why mathematicians prefer to work with convolutions rather than correlations. For instance, the extension of convolutions to infinite-dimensional inputs ($d = \infty$) and kernels ($k = \infty$) is commutative, while the extension of correlation is not. This is important also for finite d and k, because padding $\mathbf{x}$ and $\mathbf{g}$ with infinitely many zeros on both sides is a convenient way to work with convolution (and correlation, for that matter) without having to worry about what happens at boundaries.

With this type of infinite zero-padding (or with an equivalent but finite redefinition of convolution), it turns out that convolution represents polynomial multiplication, in the sense that the sequence of coefficients of the polynomial $Z(\alpha)$ resulting from the product

$$Z(\alpha) = X(\alpha)G(\alpha)$$

of polynomials $X(\alpha)$ and $G(\alpha)$ is the convolution of the sequence of coefficients of $X(\alpha)$ with the sequence of coefficients of $G(\alpha)$. This property leads in turn to important results in the theory of Fourier and Laplace transforms, used extensively in signal processing.

Layers that are made only of convolutions are called *convolutional* layers in the theory of neural networks, and a neural network that contains convolutional layers is called a *Convolutional Neural Network* (CNN). A neural network that contains *only* convolutional layers is called a *Fully Convolutional Neural Network*.

4.3 Input Padding

Convolutional kernels compute convolutions. However, it is less confusing to work with correlation than it is to work with convolutions, because there is no need to think about "flipping" the kernel. Because of this, this Section continues the discussion in terms of correlations. After all, a convolution is also a correlation, albeit with a reversed kernel.

When several convolutional layers are stacked in a cascade, with each layer taking the output of the previous one as its input, it is inconvenient that each layer is a bit smaller ($e = d - k + 1$) than the previous one. Because of this the input $\mathbf{x}$ is often padded with $p = d - e = k - 1$ zeros to produce a bigger input $\mathbf{x}'$, and the output $\mathbf{z}$ is computed as the standard correlation of $\mathbf{x}'$ and the kernel $\mathbf{g}$. In the processing of temporal signals, it would make most sense to place the p zeros at the end of the sequence (Figure 7 (a)). For images, where there is no notion of "before" or "after," a symmetric padding is used instead. One places $p_\ell = \lfloor p/2 \rfloor$ zeros on the left and $p_r = p - p_\ell$ zeros on the right. Figure 7 (b) and (c) illustrate for k odd and even, respectively.

The correlation $\mathbf{z}$ of the padded version of $\mathbf{x}$ with a kernel $\mathbf{g}$ of length k has length d, the same as that of $\mathbf{x}$. While the first p_ℓ and last p_r entries of $\mathbf{z}$ are not meaningful, the output is now equal in size to the input, and it is easier to stack several layers in a cascade. The meaningless "rim" around the output of a correlation is negligible when the input size d is large and the kernel size k is small, which is typically the case.

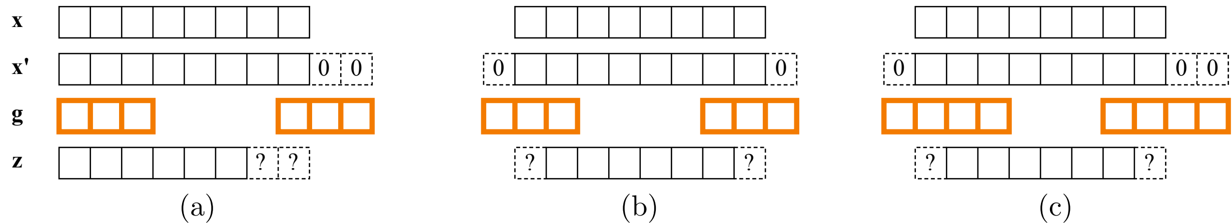

Figure 7: Input padding. In all cases, if the kernel $\mathbf{g}$ has k entries, it takes $p = k - 1$ zeros for the correlation $\mathbf{z}$ of the padded input $\mathbf{x}'$ and the kernel to have the same length as the unpadded input $\mathbf{x}$. Padding is shown with dashed contours and zeros, and the kernel (orange) is shown in its first and last valid position relative to $\mathbf{x}'$. The entries in the output $\mathbf{z}$ whose computation involves padding are shown with dashed contours and question marks. (a) In applications where sequences have a natural left-to-right ordering, such as when entries are indexed by time, padding may be added to the end of the sequence. (b) When entries of the sequences are indexed by spatial coordinates, as in images, a symmetric padding is more natural. (c) However, when k is even, the number p of padding entries is odd, and they cannot be placed with exact symmetry. In the case in this panel, $k = 4$, and the padding adds $p_\ell = 1$ zero on the left and $p_r = 2$ zeros on the right.

Correlation with input padding is called *shape-preserving* correlation, or *padded* correlation, or *'same'* correlation. To distinguish it from this style, the original, unpadded correlation is called *valid* correlation, because all of its output entries are computed form legitimate ("valid") input entries, rather than from zeros. CNN software packages usually implement both shape-preserving and valid correlation operators.

Beware: Different software packages use different terminology and conventions concerning convolution and correlation. To clarify, we consider the example of computing the `full`, `valid`, and `same` style convolution or correlation of the sequence

$$\mathbf{x} = (1, 3, 5, 2, 4) \quad \text{with kernel} \quad \mathbf{h} = (2, 3) \; .$$

All mathematicians agree (and so do these notes) that the `full` *convolution* of these two sequences is the new sequence

$$\mathbf{f} = (2, 9, 19, 19, 14, 12) \; .$$

We can compute this sequence by using the matrix formulation of convolution (note the flipped kernel in each row):

$$\mathbf{f} = H_f \mathbf{x} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 0 & 3 & 2 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ 19 \\ 19 \\ 14 \\ 12 \end{bmatrix} \; .$$

Alternatively, we can achieve the same result by first padding the input sequence $\mathbf{x}$ with a zero at the beginning and one at the end: The kernel has $n = 2$, so the necessary padding for the full convolution is $n - 1 = 1$ entry at each end. We then slide the flipped kernel $\mathbf{h}^- = (3, 2)$ in all positions where it fits entirely within $\mathbf{x}$ and compute each output entry in turn (that is, we compute the `valid` convolution of the

padded input with the kernel). In tabular form:

$$\begin{array}{ccccccc|c}
0 & 1 & 3 & 5 & 2 & 4 & 0 & \\
\hline
3 & 2 & & & & & & 2 \\
 & 3 & 2 & & & & & 9 \\
 & & 3 & 2 & & & & 19 \\
 & & & 3 & 2 & & & 19 \\
 & & & & 3 & 2 & & 14 \\
 & & & & & 3 & 2 & 12
\end{array}$$

Similarly, all mathematicians (and these notes) agree that the `full` *correlation* of $\mathbf{x}$ with $\mathbf{h}$ is the sequence

$$\boldsymbol{\varphi} = (3, 11, 21, 16, 16, 8) \; .$$

This can be computed the same ways as above, but without flipping the kernel.

However, the `PyTorch` deep learning package calls correlations "convolutions." For instance, the commands

```
import torch
x = torch.tensor([[1, 3, 5, 2, 4]])
h = torch.tensor([[[2, 3]]])
torch.nn.functional.conv1d(x, h, padding=1)
```

produce output

```
tensor([[ 3, 11, 21, 16, 16,  8]])
```

which is the *correlation* $\boldsymbol{\varphi}$, even if the operation is called `conv1d` (don't worry about all the extra square brackets above, they are needed because of things called "batches" and "channels," but they are irrelevant here). If you wanted an actual convolution from `PyTorch` you would have to flip the kernel explicitly:

```
torch.nn.functional.conv1d(x, torch.flip(h, dims=(2,)), padding=1)
```

This indeed produces the convolution $\mathbf{f}$:

```
tensor([[ 2,  9, 19, 19, 14, 12]])
```

Note also that there is no option `padding='full'` in `PyTorch`. You need to figure out the actual amount of padding yourself, and the software adds that amount at both ends. However, you may use `padding='valid'` or `padding='same'` with `conv1d`. This is an odd choice, because (i) the default value for `padding` is zero, which is already equivalent to `padding='valid'`; and(ii) the `padding='full'` option would be very easy to implement, since the amount of padding is simply one less than the length of the kernel.

Another discrepancy among packages concerns how to split the padding between beginning and end of the input when the kernel $\mathbf{h}$ has an even number n of entries, as in the example above. In that case, the padding is $k = n//2$ at one end and the remaining $n - k$ at the other, but different packages do it differently.

Specifically, MATLAB pads at the end, which, if you think about it, is the same as removing rows from the *top* of H_f. For our running example, the MATLAB command

```
conv([1 3 5 2 4], [2 3], 'same')
```

yields

```
9     19     19     14     12
```

which we can recompute as follows

$$\mathbf{s} = H_s \mathbf{x} = \begin{bmatrix} 3 & 2 & 0 & 0 & 0 \\ 0 & 3 & 2 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 2 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 9 \\ 19 \\ 19 \\ 14 \\ 12 \end{bmatrix}$$

or, in tabular form:

1	3	5	2	4	0	
3	2					9
	3	2				19
		3	2			19
			3	2		14
				3	2	12

On the other hand, the Python `numpy` package pads at the beginning (or, equivalently, truncates the full matrix at the bottom). Specifically, the commands

```
import numpy as np
np.convolve(np.array([1, 3, 5, 2, 4]), np.array([2, 3]), 'same')
```

yield

```
array([ 2,  9, 19, 19, 14])
```

which we can recompute as follows:

$$\mathbf{s}' = H_s'\mathbf{x} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 3 & 2 & 0 & 0 & 0 \\ 0 & 3 & 2 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 0 & 0 & 3 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \\ 5 \\ 2 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 9 \\ 19 \\ 19 \\ 14 \end{bmatrix}$$

or, in tabular form:

0	1	3	5	2	4	
3	2					2
	3	2				9
		3	2			19
			3	2		19
				3	2	14

Finally, `PyTorch` agrees with MATLAB and disagrees with `numpy` as far as padding or truncating for the `same` option. If we manually flip the kernel to simulate convolution, the command

```
torch.nn.functional.conv1d(x, torch.flip(h, dims=(2,)), padding='same')
```

yields

```
tensor([[ 9, 19, 19, 14, 12]])
```

which is equivalent to the output from MATLAB.

4.4 Two-Dimensional Correlation

The concept of correlation can be extended in straightforward fashion to signals defined in any number of dimensions, rather than just one. This extension is now examined in the two-dimensional case, which is most important for images. Instead of thinking of a kernel as a sound clip (drumbeat), think of it now as a small image detail (perhaps an eye).

If the input image X is an array with $d_1 \times d_2$ entries x_{ij} and the kernel G is an array with $k_1 \times k_2$ entries g_{ab}, then the entries z_{ij} of the valid correlation Z of X and G are defined by an immediate extension of equation 1:

$$z_{ij} = \sum_{a=0}^{k_1-1} \sum_{b=0}^{k_2-1} g_{ab}\, x_{i+a,j+b} \quad \text{for} \quad i = 0, \ldots, e_1 - 1 = d_1 - k_1 \quad \text{and} \quad j = 0, \ldots, e_2 - 1 = d_2 - k_2 \; . \quad (2)$$

Figure 8, top, shows the sliding-window view of the two-dimensional correlation for the valid correlation of a $d_1 \times d_2 = 4 \times 6$ image X with a $k_1 \times k_2 = 3 \times 2$ kernel G. The output Z has $e_1 \times e_2 = (4 - 3 + 1) \times (6 - 2 + 1) = 2 \times 5$ entries.

What used to be the "matrix" view can now be generalized in two different ways to the two-dimensional case. The more natural is a *tensor view*, in which the array V that represents the transformation from X to Z is four-dimensional, and has entries V_{ijab}. We can then write

$$z_{ij} = \sum_{a=0}^{d_1-1} \sum_{b=0}^{d_2-1} v_{ijab} x_{ab} \quad \text{for} \quad i = 0, \ldots, e_1 - 1 \quad \text{and} \quad j = 0, \ldots, e_2 - 1 \; , \tag{3}$$

and comparison with equation 2 shows that

$$v_{ijab} = g_{ab} \quad \text{for} \quad i = 0, \ldots, e_1 - 1 \quad \text{and} \quad j = 0, \ldots, e_2 - 1 \; . \tag{4}$$

Thus equation 3 expresses an *arbitrary* linear transformation between a two-dimensional input and a two-dimensional output. With the constraints in equation 4, on the other hand, this linear transformation specializes to a correlation.

The second way to generalize the "matrix view" of correlation to the two-dimensional case is to first "flatten" input X into a vector $\mathbf{x} \in \mathbb{R}^{d_1 d_2}$ and output Z into a vector $\mathbf{z} \in \mathbb{R}^{e_1 e_2}$ and then give the corresponding matrix V_f. Figure 8, middle, illustrates.

Padding works in each dimension of a two-dimensional correlation the same way as for one-dimensional correlation. If the input image is $d_1 \times d_2$ and the kernel is $k_1 \times k_2$, then padding takes

$$p_m = k_m - 1$$

zeros in dimension m for $m = 1, 2$. For the example in Figure 8, the image is 4×6 and the kernel is 3×2, so that

$$p_1 = k_1 - 1 = 2 \quad \text{and} \quad p_2 = k_2 - 1 = 1$$

padding zeros in the two dimensions. With symmetric padding, this means adding a row above, a row below, and a column to the right of the input image.

4.5 Stride

In a standard correlation operation, the kernel is slid over the entire input. To this end, it is moved by one pixel to the right in each row. When that row of the output is computed, the kernel is moved to the beginning of the next row, and the process is repeated.

If the kernel is not too small, the output entry z_{ij} is not very different from the output entry $z_{i,j+1}$ or the output entry $z_{i+1,j}$, because images often vary slowly as a function of image coordinates. To reduce the resulting redundancy in the output, correlations are often computed with a *stride* s_m grater than one. That is, after z_{ij} for some i and j has been computed, the kernel is translated by s_1 pixels horizontally, rather than just one. Once row i has been completed in this fashion, the kernel is moved down by s_2 rows, rather than just one. In this way, the output has size roughly equal to $d_1/s_1 \times d_2/s_2$. Padding can be used so that this size holds exactly, if the two fractions are integer, or approximately otherwise.

$$
V_f = \begin{bmatrix}
g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & g_{00} & g_{01} & 0 & 0 & 0 & 0 & g_{10} & g_{11} & 0 & 0 & 0 & 0 & g_{20} & g_{21}
\end{bmatrix}
$$

Figure 8: Top: Sliding-window view of the valid correlation of a 4×6 image X (black) with a 3×2 kernel G (orange). Middle: The “flattened matrix” corresponding to this correlation. Entries in each array are assumed to be listed in row-major order, that, is, with the column index changing fastest. Bottom: Zero padding needed for the output of the correlation of X and G to have the same size as X. The kernel (orange) is shown in its first and last position. With padded convolution, the flattened matrix V_f is square (24×24 in the example).

5 The Structure of CNNs

The notions of locality, reuse, and resilience to spatial variations discussed earlier suggest the following structure for a neural-net layer [6, 5].

- Think of the input $\mathbf{x}$ as a two-dimensional array, one entry per pixel, rather than a vector, so that the notion of locality is more readily expressed.
- Group the activations a_i (the entries of $\mathbf{a}$) into m *maps*: each map takes care of one type of pattern through a separate correlation kernel with a small *support* $k_1 \times k_2$. A pattern kernel with a small support is also called a *feature*, so the m maps are called *feature maps*. The (common) activation function ρ is then applied to each feature map, entry by entry.
- Reduce the size of each feature map by *max-pooling*. Specifically, square supports are defined in each feature map in turn, with a size and stride that is common to all the feature maps. A new, smaller feature map is then computed whose values are each the maximum value in its support.

  In addition to reducing the size of the feature maps in the output from the layer, max-pooling makes the output of the layer somewhat less sensitive to the exact location of the features in the image. For instance, with a 3×3 support for max-pooling and a stride of 3 (no overlap between pools), the output of the maximum is oblivious to which of those 3^2 activations produced the final output from the layer. In other words, max-pooling achieves some degree of *translation-invariance*. If the same is done in every layer of a deep network, the amounts of invariance add up.

The *convolutional organization* described above is illustrated in Figure 9 for the input layer of a network from the literature [5]. This layer includes max-pooling. As a result of this structure, the number of distinct parameters in the layer drops dramatically when compared to that for fully connected layers. In a *fully connected* layer, each input scalar is connected through a trainable gain to each output scalar, as we have seen for generic layers so far. With a $n \times n \times 3$ array (a square color image) as input and m output feature maps of dimension $n \times n$ as output there would be $3mn^4$ gains and mn^2 bias terms (one bias per output scalar).

With m convolution kernels (m feature maps), on the other hand, the number of parameters is $(3k^2 + 1)m$ if all the kernels are $k \times k$, and if the single bias term added for each feature map is counted as well. For color $n \times n$ images and m feature maps of size $n \times n$, the count drops from $O(mn^4)$ weights for a fully connected layer to $m(3k^2 + 1)$ weights for a convolutional one.

For instance [5], for a $224 \times 224 \times 3$-pixel color image and 96 maps with an 11×11 receptive field, the drop is from about 10^{10} parameters to a mere $96 \times (11^2 + 1) \approx 10^4$ parameters.

The original paper [5] does not describe the padding policy used for the convolutions. With the "same" style in the two image dimensions and a stride of 4 in each image dimension, each feature map should have width and height equal to $\lfloor 224/4 \rfloor = 56$. However, the paper reports feature maps that are 55×55 pixels each.[5] Max-pooling uses 3×3-pixel receptive fields and a stride of 2 pixels, and produces output maps of size 27×27 pixels.

[5] A "valid" convolution with stride 4 would yield a feature map smaller than 55×55, and a "full" convolution would yield a feature map greater than 56×56. We cannot explain the discrepancy of one pixel between our calculation and the paper's.

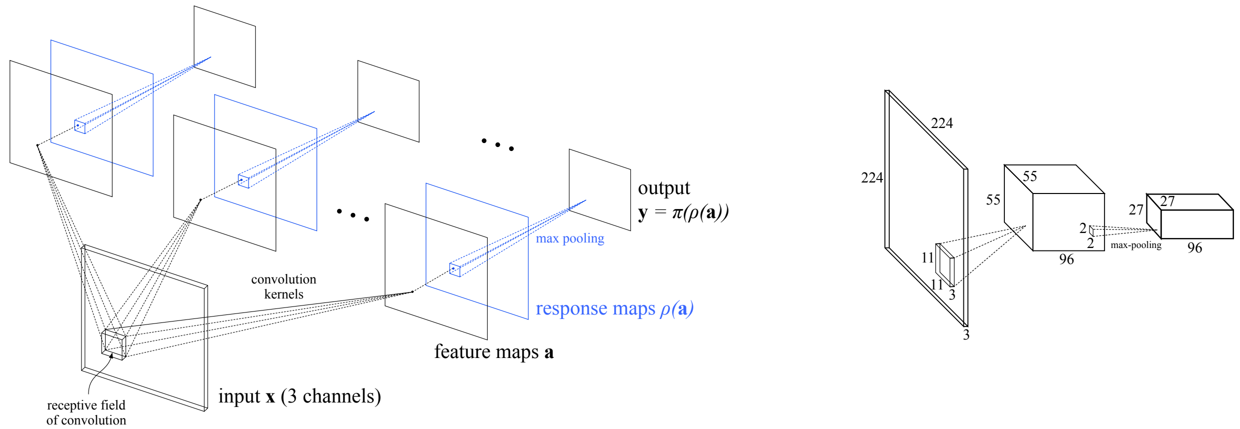

Figure 9: (Left) The structure of a convolutional neural-net layer for a color image with three channels (red, green, blue). (Right) In the literature, neural nets with many layers are drawn with each layer shown in a more compact way than on the left, although there is no standard format. Typically, the maps are stacked in a block, as shown here, rather than drawn side-to-side. Sometimes, max-pooling is only mentioned and not shown explicitly.

On the other hand, each 11×11 kernel must produce exactly one output feature map, so the convolution style in the channel dimension must be the "valid" style. Feature maps are also called *output channels* in the literature.

The set of activation maps computed by the 96 kernels is a $55 \times 55 \times 96$ block, and max pooling with stride 2 is applied to each of the 96 slices in this block to produce a $27 \times 27 \times 96$ output block from this layer. Thus, for the layer in the Figure (including max-pooling), the output dimensionality is $e = 27^2 \times 96 = 69{,}984$, a bit less than half of the input dimensionality $d = 224^2 \times 3 = 150{,}528$. On the other hand, the map *resolution* decreases more than eightfold, from 224 to 27 pixels on each side. The representation of the image has become more abstract, changing from a pure pixel-by-pixel list of its colors to a coarser map of how much each of 96 features is present at each (coarse) location in the image and in each color channel.

6 Deep Convolutional Neural Nets

The architecture of a neural-net layer embodies the principles of feature reuse, locality, and translation-invariance. *Deep* Convolutional Neural Nets (CNNs) are CNNs with many layers, and reflect the principle of hierarchy. After several convolutional layers, deep CNNs typically add one or a few fully-connected layers, that is, layers where the weight matrix W is dense. The reasons for doing so are somewhat mixed and not entirely compelling, but are nonetheless plausible: Far away from the input, spatial location is both partially lost and relatively irrelevant to, say, recognition, so the local supports of CNNs are no longer useful. In addition, signals in late stages of a deep net have relatively low dimensionality, and one can then better afford the greater representational flexibility that a fully-connected layer carries.

The output from a deep CNN is fed to a computation that depends on the purpose of the net. For regression, for instance, the outputs may be used as they are. For classification, one could use the outputs as inputs to a support vector machine or random forest. More commonly, the output

stage is a *softmax* function,

$$\mathbf{z} = \sigma(\mathbf{y}) = \frac{\exp(\mathbf{y})}{\mathbf{1}^T \exp(\mathbf{y})}$$

where $\mathbf{1}$ is a column vector of ones. As we saw in an erlier note, the exponential makes all quantities positive, and normalization makes sure the entries of $\mathbf{z}$ add up to 1. In this way, the entries of the softmax output can be viewed as *normalized scores* for each of the categories, and the result of classification is then class

$$h(\mathbf{x}) = \arg\max_i z_i \; .$$

References

[1] C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

[2] L. Deng and D. Yu. *Deep Learning: Methods and Applications*, volume 7(3-4) of *Foundations and Trends in Signal Processing.* Now Publishers, 2014.

[3] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning.* The MIT Press, Cambridge, MA, 2016.

[4] V. Y. Kreinovic. Arbitrary nonlinearity is sufficient to represent all functions by neural networks: a theorem. *Neural Networks*, 4:381–383, 1991.

[5] A. Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1106–1114, 2012.

[6] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[7] S. Sonoda and N. Murata. Neural network with unbounded activations is universal approximator. Technical Report 1505.3654 [cs.NE], arXiv, 2015.