# Chapter 9

# Convolutional Networks

*Convolutional networks* (also known as *convolutional neural networks* or *CNNs*) are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. Convolutional networks have been tremendously successful in practical applications (the specifics of several of these applications will be explained in Chapter 12.2.2). The name "convolutional neural network" indicates that the network employs a mathematical operation called *convolution*. Convolution is a specialized kind of linear operation. **Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.**

KEY
IDEA

In this chapter, we will first describe what convolution is. Next, we will explain the motivation behind using convolution in a neural network. We will then describe an operation called *pooling*, which almost all convolutional networks employ. Usually, the operation used in a convolutional neural network does not correspond precisely to the definition of convolution as used in other fields such as engineering or pure mathematics. We will describe several variants on the convolution function that are widely used in practice for neural networks. We will also show how convolution may be applied to many kinds of data, with different numbers of dimensions. We then discuss means of making convolution more efficient. Convolutional networks standard out as an example of neuroscientific principles influencing deep learning. We will discuss these neuroscientific principles, then conclude with comments about the role convolutional networks have played in the history of deep learning.

# 9.1 The Convolution Operation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, let's start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the spaceship at time $t$. Both $x$ and $t$ are real-valued, i.e., we can get a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average together several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where $a$ is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function $s$ providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called *convolution*. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In our example, $w$ needs to be a valid probability density function, or the output is not a weighted average. Also, $w$ needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example though. In general, convolution is defined for any functions for which the above integral is defined, and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function $x$) to the convolution is often referred to as the *input* and the second argument (in this example, the function $w$) as the *kernel*. The output is sometimes referred to as the *feature map*.

In our example, the idea of a laser sensor that can provide measurements at every instant in time is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides one measurement once per second. $t$ can then take on only integer values. If we now assume that $x$ and $w$ are defined only on integer $t$, we can

define the discrete convolution:

$$s[t] = (x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t-a]$$

TODO: synch w/Yoshua and Aaron about how to handle this kind of indexing. Add an integer-domain function line to notation.tex?

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of learn-able parameters. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image $I$ as our input, we probably also want to use a two-dimensional kernel $K$:

$$s[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n}I[m,n]K[i-m,j-n]$$

Note that convolution is commutative, meaning we can equivalently write:

$$s[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n}I[i-m,j-n]K[m,n]$$

Usually the latter view is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of $m$ and $n$.

While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the *cross-correlation*, which is the same as convolution but without flipping the kernel:

$$s[i,j] = (I * K)[i,j] = \sum_{m}\sum_{n}I[i+m,j+n]K[m,n]$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we will follow this convention of calling both operations convolution, and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant.

See Fig. 9.1 for an example of convolution (without kernel flipping) applied to a 2-D tensor.

Input

Kernel

Output

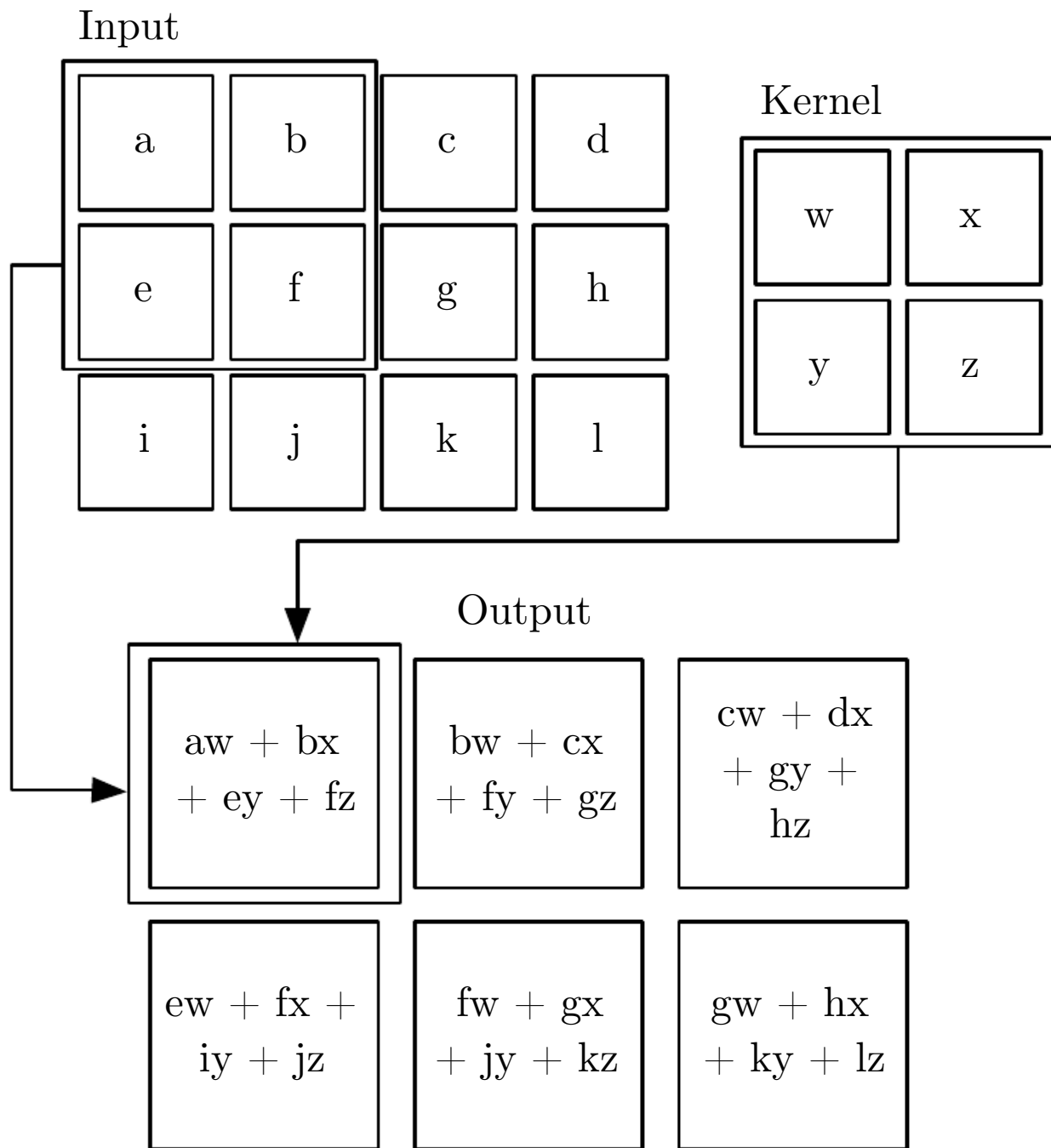| | | |
|---|---|---|
| aw + bx + ey + fz | bw + cx + fy + gz | cw + dx + gy + hz |
| ew + fx + iy + jz | fw + gx + jy + kz | gw + hx + ky + lz |

Figure 9.1: An example of 2-D convolution without kernel-flipping. In this case we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

Discrete convolution can be viewed as multiplication by a matrix. However, the matrix has several entries constrained to be equal to other entries. For example, for univariate discrete convolution, each row of the matrix is constrained to be equal to the row above shifted by one element. This is known as a *Toeplitz matrix*. In two dimensions, a *doubly block circulant matrix* corresponds to convolution. In addition to these constraints that several elements be equal to each other, convolution usually corresponds to a very sparse matrix (a matrix whose entries are mostly equal to zero). This is because the kernel is usually much smaller than the input image. Viewing convolution as matrix multiplication usually does not help to implement convolution operations, but it is useful for understanding and designing neural networks. Any neural network algorithm that works with matrix multiplication and does not depend on specific properties of the matrix structure should work with convolution, without requiring any further changes to the neural network. Typical convolutional neural networks do make use of further specializations in order to deal with large inputs efficiently, but these are not strictly necessary from a theoretical perspective.

## 9.2  Motivation

Convolution leverages three important ideas that can help improve a machine learning system: *sparse interactions*, *parameter sharing*, and *equivariant representations*. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

Traditional neural network layers use a matrix multiplication to describe the interaction between each input unit and each output unit. This means every output unit interacts with every input unit. Convolutional networks, however, typically have *sparse interactions* (also referred to as *sparse connectivity* or *sparse weights*). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations. These improvements in efficiency are usually quite large. If there are $m$ inputs and $n$ outputs, then matrix multiplication requires $m \times n$ parameters and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to $k$, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping $k$ several orders of
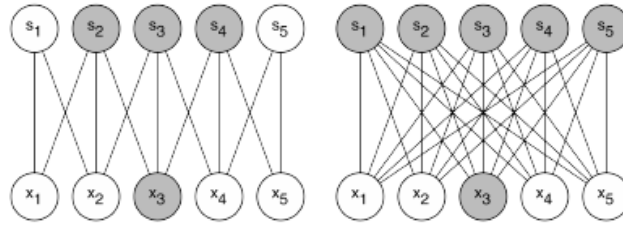
Figure 9.2: *Sparse connectivity, viewed from below:* We highlight one input unit, $X_3$, and also highlight the output units in **S** that are affected by this unit. *(Left)* When **S** is formed by convolution with a kernel of width 3, only three outputs are affected by $X_3$. *(Right)* When **S** is formed by matrix multiplication, connectivity is no longer sparse, so all of the outputs are affected by $X_3$. TODO: make sure fig uses latest notation
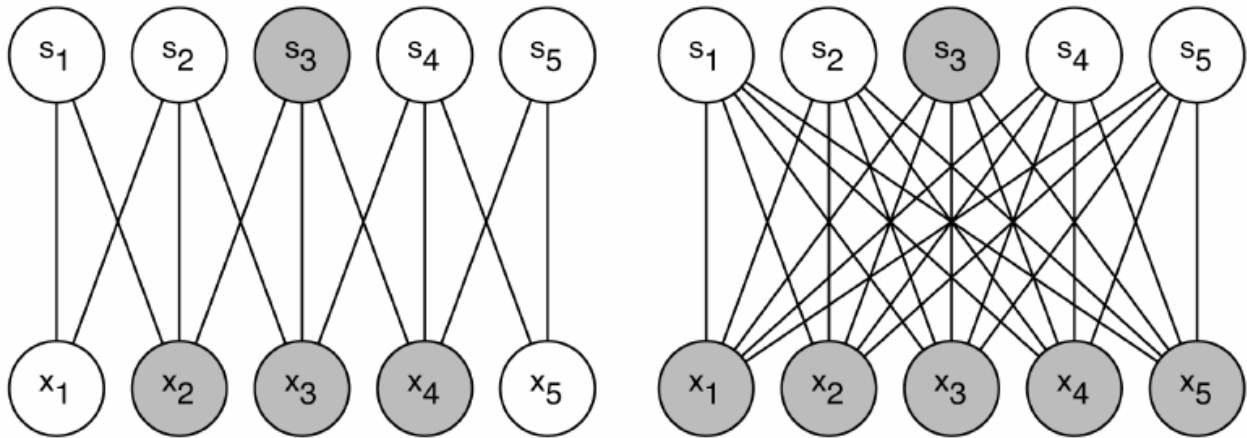


Figure 9.3: *Sparse connectivity, viewed from above:* We highlight one output unit, $S_3$, and also highlight the input units in **X** that affect this unit. These units are known as the *receptive field* of $S_3$. *(Left)* When **S** is formed by convolution with a kernel of width 3, only three inputs affect $S_3$. *(Right)* When **S** is formed by matrix multiplication, connectivity is no longer sparse, so all of the inputs affect $S_3$. TODO: make sure fig uses latest notation

magnitude smaller than $m$. For graphical demonstrations of sparse connectivity, see Fig. 9.2 and Fig. 9.3. In a deep convolutional network, units in the deeper layers may *indirectly* interact with a larger portion of the input, as shown in Fig. 9.4. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

*Parameter sharing* refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input, and then never revisited. As a synonym for parameter sharing, one can say that a network has *tied weights*, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In
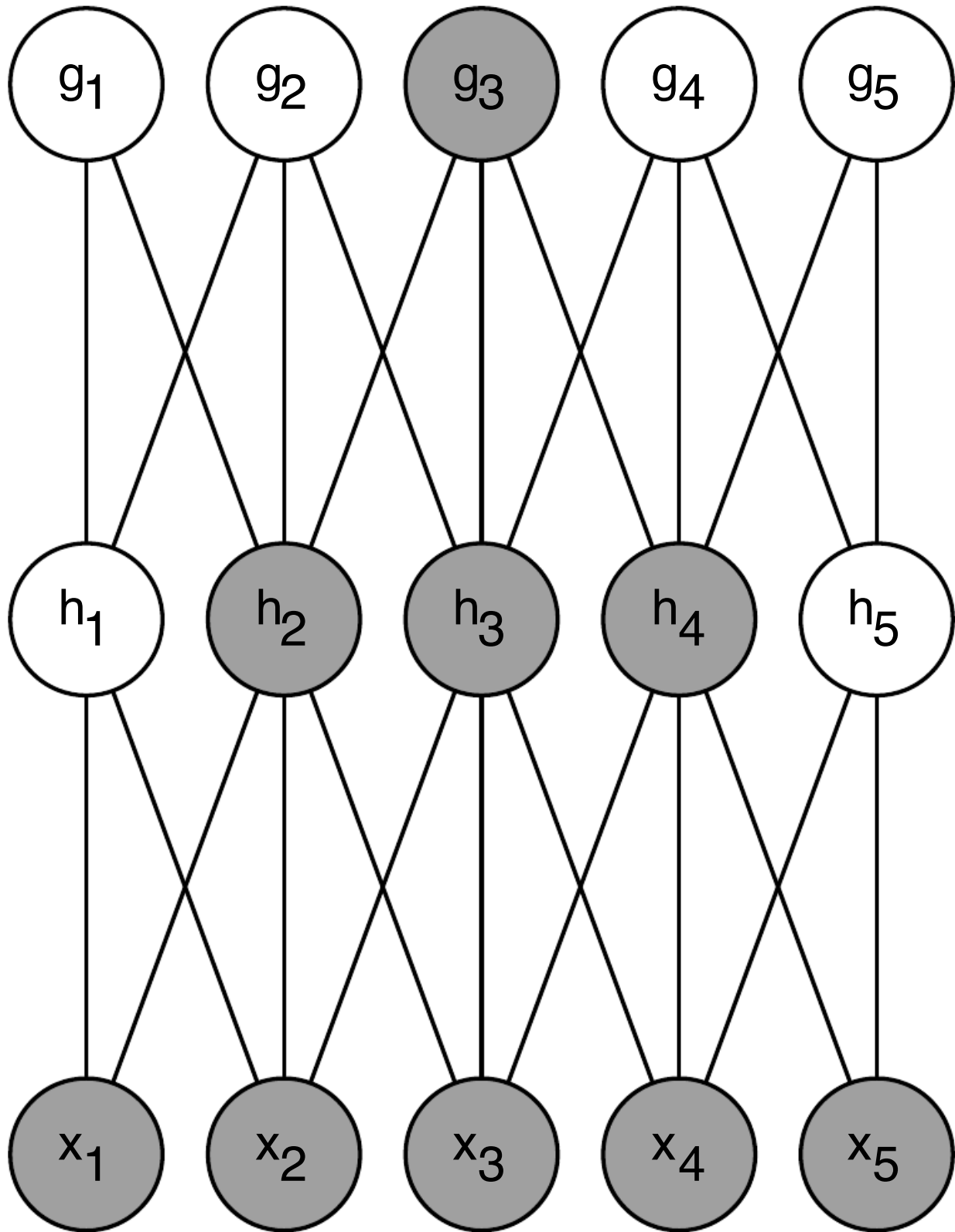
Figure 9.4: The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution or pooling. This means that even though *direct* connections in a convolutional net are very sparse, units in the deeper layers can be *indirectly* connected to all or most of the input image.
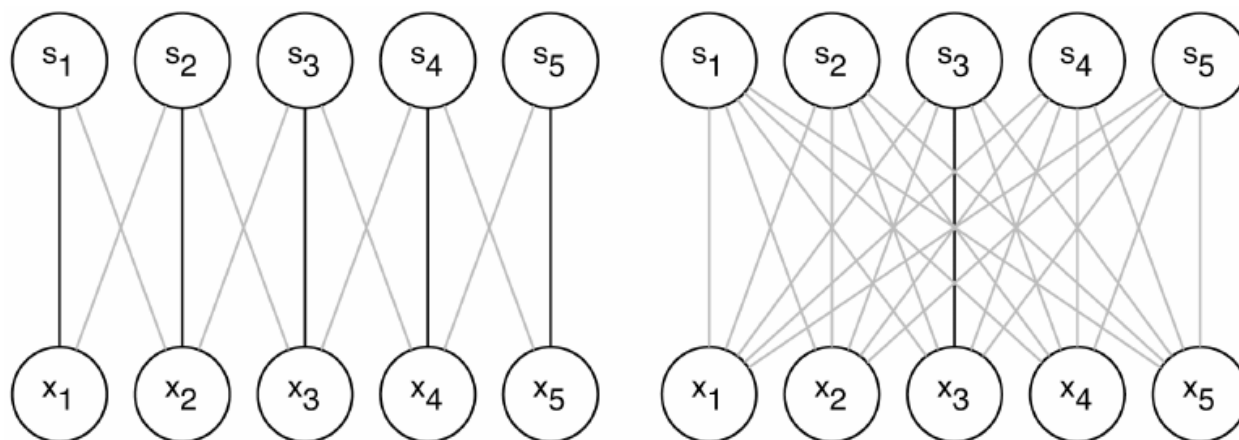
Figure 9.5: *Parameter sharing:* We highlight the connections that use a particular parameter in two different models. *(Left)* We highlight uses of the central element of a 3-element kernel in a convolutional model. Due to parameter sharing, this single parameter is used at all input locations. *(Right)* We highlight the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing so the parameter is used only once.

a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation–it is still $O(k \times n)$–but it does further reduce the storage requirements of the model to $k$ parameters. Recall that $k$ is usual several orders of magnitude less than $m$. Since $m$ and $n$ are usually roughly the same size, $k$ is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see Fig. 9.5.

As an example of both of these first two principles in action, Fig. 9.6 shows how sparse connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image.

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called *equivariance* to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g$ if $f(g(x)) = g(f(x))$. In the case of convolution, if we let $g$ be any function that translate the input, i.e., shifts it, then the convolution function is equivariant to $g$. For example, define $g(x)$ such that for all $i$, $g(x)[i] = x[i-1]$. This shifts every element of $x$ one unit to the right. If we apply this transformation to $x$, then apply convolution,

the result will be the same as if we applied convolution to $x$, then applied the transformation to the output. When processing time series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later in time. Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that same local function is useful everywhere in the input. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network, and an edge looks the same regardless of where it appears in the image. This property is not always useful. For example, if we want to recognize a face, some portion of the network needs to vary with spatial location, because the top of a face does not look the same as the bottom of a face–the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin.

Note that convolution is not equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. We discuss this further in section 9.8.

## 9.3   Pooling

A typical layer of a convolutional network consists of three stages (see Fig. 9.7). In the first stage, the layer performs several convolutions in parallel to produce a set of presynaptic activations. In the second stage, each presynaptic activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the *detector* stage. In the third stage, we use a *pooling function* to modify the output of the layer further.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the *max pooling* operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

In all cases, pooling helps to make the representation become *invariant* to small translations of the input. This means that if we translate the input by a small amount, the values of most of the pooled outputs do not change. See
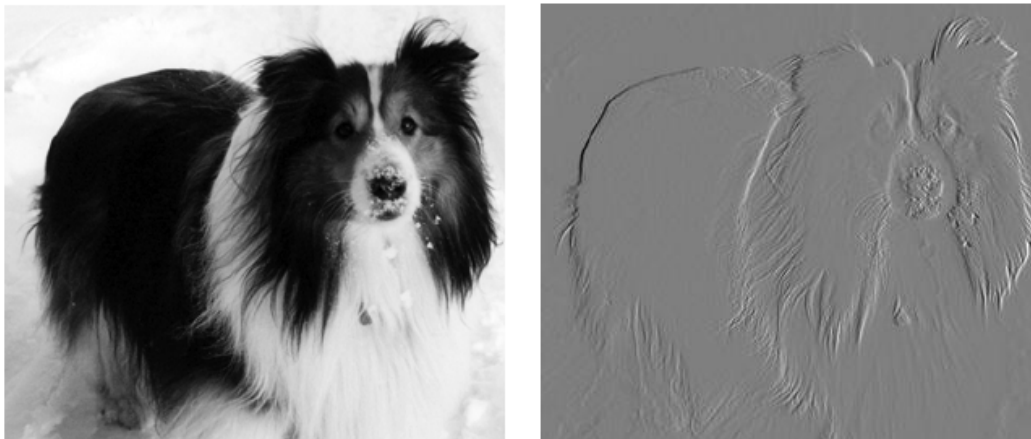
Figure 9.6: *Efficiency of edge detection.* The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all of the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing 2 elements, and requires $319 \times 280 \times 3 = 267,960$ floating point operations (two multiplications and one addition per output pixel) to compute. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over 8 billion, entries in the matrix, making convolution 4 billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over 16 billion floating point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. Convolution is an extremely efficient way of describing transformations that apply the same linear transformation of a small, local region across the entire input. (Photo credit: Paula Goodfellow)

Complex layer terminology | Simple layer terminology

```
          Next layer                      Next layer

    Convolutional Layer

        Pooling stage                    Pooling layer

       Detector stage:                  Detector layer:
         Nonlinearity                      Nonlinearity
     e.g., rectified linear           e.g., rectified linear

      Convolution stage:              Convolution layer:
        Affine transform                Affine transform

         Input to layer                  Input to layers
```
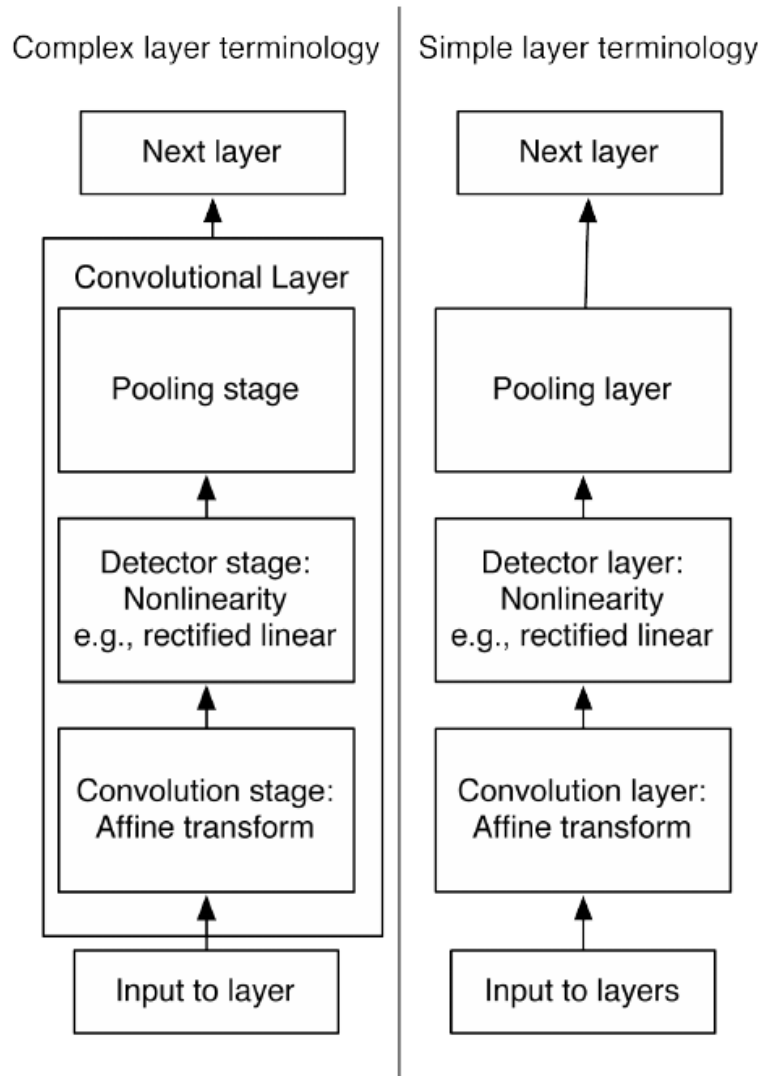
Figure 9.7: The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. *Left)* In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages." In this terminology, there is a one-to-one mapping between kernel tensors and network layers. In this book we generally use this terminology. *Right)* In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters.
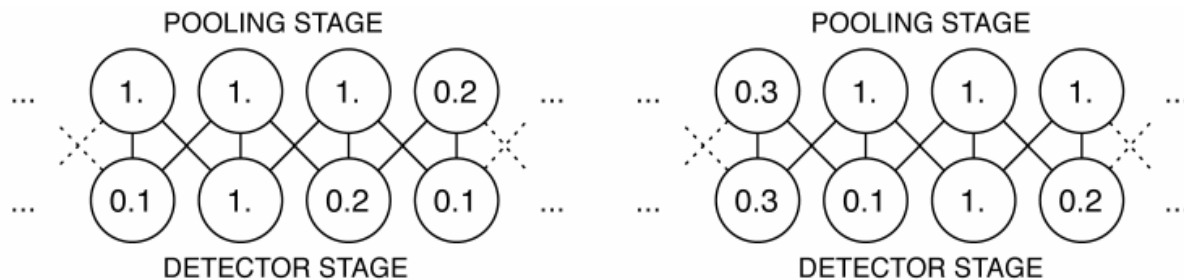
Figure 9.8: *Max pooling introduces invariance.* Left: A view of the middle of the output of a convolutional layer. The bottom row shows outputs of the nonlinearity. The top row shows the outputs of max pooling, with a stride of 1 between pooling regions and a pooling region width of 3. Right: A view of the same network, after the input has been shifted to the right by 1 pixel. Every value in the bottom row has changed, but only half of the values in the top row have changed, because the max pooling units are only sensitive to the maximum value in the neighborhood, not its exact location.

Fig. 9.8 for an example of how this works. **Invariance to local translation can be a very useful property if we care more about whether some feature is present than exactly where it is.** For example, when determining whether an image contains a face, we need not know the location of the eyes with pixel-perfect accuracy, we just need to know that there is an eye on the left side of the face and an eye on the right side of the face. In other contexts, it is more important to preserve the location of a feature. For example, if we want to find a corner defined by two edges meeting at a specific orientation, we need to preserve the location of the edges well enough to test whether they meet. KEY IDEA

The use of pooling can be viewed as adding an infinitely strong prior that the function the layer learns must be invariant to small translations. When this assumption is correct, it can greatly improve the statistical efficiency of the network.

Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to (see Fig. 9.9).

Because pooling summarizes the responses over a whole neighborhood, it is possible to use fewer pooling units than detector units, by reporting summary statistics for pooling regions spaced $k$ pixels apart rather than 1 pixel apart. See Fig. 9.10 for an example. This improves the computational efficiency of the network because the next layer has roughly $k$ times fewer inputs to process. When the number of parameters in the next layer is a function of its input size (such as when the next layer is fully connected and based on matrix multiplication) this reduction in the input size can also result in improved statistical efficiency and reduced memory requirements for storing the parameters.
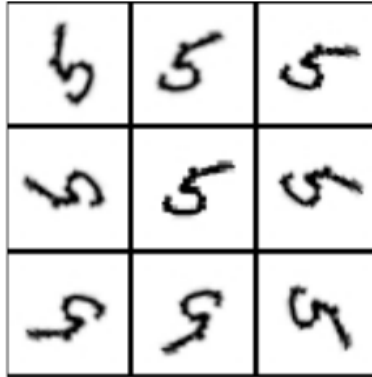
TODO: figure resembling `http://deeplearning.net/tutorial/lenet.html#the-full-m`

Figure 9.9: *Example of learned invariances:* If each of these filters drive units that appear in the same max-pooling region, then the pooling unit will detect "5"s in any rotation. By learning to have each filter be a different rotation of the "5" template, this pooling unit has learned to be invariant to rotation. This is in contrast to translation invariance, which is usually achieved by hard-coding the net to pool over shifted versions of a single learned filter.
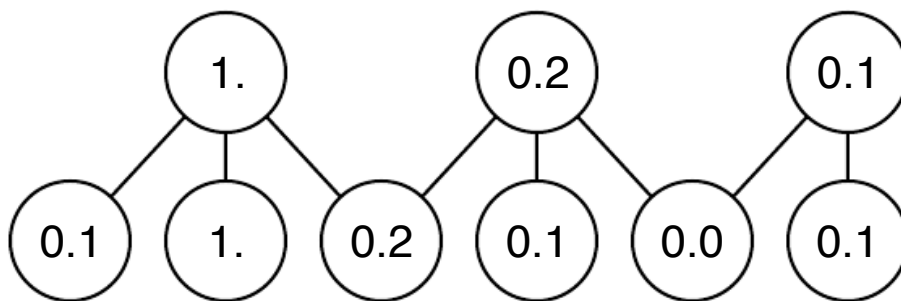


Figure 9.10: *Pooling with downsampling.* Here we use max-pooling with a pool width of 3 and a stride between pools of 2. This reduces the representation size by a factor of 2, which reduces the computational and statistical burden on the next layer. Note that the final pool has a smaller size, but must be included if we do not want to ignore some of the detector units.

e.g. show a representative example of a net with multiple layers, different numbers of filters at each layer, different spatial sizes as you go deeper

For many tasks, pooling is essential for handling inputs of varying size. For example, if we want to classify images of variable size, the input to the classification layer must have a fixed size. This is usually accomplished by varying the size of and offset between pooling regions so that the classification layer always receives the same number of summary statistics regardless of the input size. For example, the final pooling layer of the network may be defined to output four sets of summary statistics, one for each quadrant of an image, regardless of the image size. TODO: add figure showing a classifier network with a fully connected layer, and then one with global average pooling.

Some theoretical work gives guidance as to which kinds of pooling one should use in various situations (Boureau *et al.*, 2010). It is also possible to dynamically pool features together, for example, by running a clustering algorithm on the locations of interesting features (Boureau *et al.*, 2011). This approach yields a different set of pooling regions for each image. Another approach is to *learn* a single pooling structure that is then applied to all images (Jia *et al.*, 2012).

Pooling can complicate some kinds of neural network architectures that use top-down information, such as Boltzmann machines and autoencoders. These issues will be discussed further when we present these types of networks. Pooling in convolutional Boltzmann machines is presented in Chapter 20.7. The inverse-like operations on pooling units needed in some differentiable networks will be covered in Chapter 20.9.6.

## 9.4 Convolution and Pooling as an Infinitely Strong Prior

Recall the concept of a *prior probability distribution* from Chapter 5.3. This is a probability distribution over the parameters of a model that encodes our beliefs about what models are reasonable, before we have seen any data.

Priors can be considered weak or strong depending on how concentrated the probability density in the prior is. A weak prior is a prior distribution with high entropy, such a Gaussian distribution with high variance. Such a prior allows the data to move the parameters more or less freely. A strong prior has very low entropy, such as a Gaussian distribution with low variance. Such a prior plays a more active role in determining where the parameters end up.

An infinitely strong prior places zero probability on some parameters and says that these parameter values are completely forbidden, regardless of how much support the data gives to those values.

We can imagine a convolutional net as being similar to a fully connected net,

but with an infinitely strong prior over its weights. This infinitely strong prior says that the weights for one hidden unit must be identical to the weights of its neighbor, but shifted in space. The prior also says that the weights must be zero, except for in the small, spatially contiguous receptive field assigned to that hidden unit. Overall, we can think of the use of convolution as introducing an infinitely strong prior probability distribution over the parameters of a layer. This prior says that the function the layer should learn contains only local interactions and is equivariant to translation. Likewise, the use of pooling is in infinitely strong prior that each unit should be invariant to small translations.

Of course, implementing a convolutional net as a fully connected net with an infinitely strong prior would be extremely computationally wasteful. But thinking of a convolutional net as a fully connected net with an infinitely strong prior can give us some insights into how convolutional nets work.

One key insight is that convolution and pooling can cause underfitting. Like any prior, convolution and pooling are only useful when the assumptions made by the prior are reasonably accurate. If a task relies on preserving precision spatial information, then using pooling on all features can cause underfitting. (Some convolution network architectures (Szegedy *et al.*, 2014a) are designed to use pooling on some channels but not on other channels, in order to get both highly invariant features and features that will not underfit when the translation invariance prior is incorrect) When a task involves incorporating information from very distant locations in the input, then the prior imposed by convolution may be inappropriate.

Another key insight from this view is that we should only compare convolutional models to other convolutional models in benchmarks of statistical learning performance. Models that do not use convolution would be able to learn even if we permuted all of the pixels in the image. For many image datasets, there are separate benchmarks for models that are *permutation invariant* and must discover the concept of topology via learning, and models that have the knowledge of spatial relationships hard-coded into them by their designer.

## 9.5 Variants of the Basic Convolution Function

When discussing convolution in the context of neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature. The functions used in practice differ slightly. Here we describe these differences in detail, and highlight some useful properties of the functions used in neural networks.

First, when we refer to convolution in the context of neural networks, we usually actually mean an operation that consists of many applications of convolution

in parallel. This is because convolution with a single kernel can only extract one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations.

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green, and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. (Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch)

Note that because convolutional networks usually use multi-channel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel-flipping is used. These multi-channel operations are only commutative if each operation has the same number of output channels as input channels.

Assume we have a 4-D kernel tensor $\mathbf{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel $i$ of the output and a unit in channel $j$ of the input, with an offset of $k$ rows and $l$ columns between the output unit and the input unit. Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel $i$ at row $j$ and column $k$. Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$. If $\mathbf{Z}$ is produced by convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n}$$

where the summation over $l$, $m$, and $n$ is over all values for which the tensor indexing operations inside the summation is valid.

We may also want to skip over some positions of the kernel in order to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as downsampling the output of the full convolution function. If we want to sample only every $s$ pixels in each direction in the output, then we can defined a downsampled convolution function $c$ such that:

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[ V_{l,j\times s+m,k\times s+n} K_{i,l,m,n} \right]. \tag{9.1}$$

We refer to $s$ as the *stride* of this downsampled convolution. It is also possible to define a separate stride for each direction of motion. TODO add a figure for

this

One essential feature of any convolutional network implementation is the ability to implicitly zero-pad the input **V** in order to make it wider. Without this feature, the width of the representation shrinks by the kernel width - 1 at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels–both scenarios that significantly limit the expressive power of the network. See Fig. 9.11 for an example.

Three special cases of the zero-padding setting are worth mentioning. One is the extreme case in which no zero-padding is used whatsoever, and the convolution kernel is only allowed to visit positions where the entire kernel is contained entirely within the image. In MATLAB terminology, this is called *valid* convolution. In this case, all pixels in the output are a function of the same number of pixels in the input, so the behavior of an output pixel is somewhat more regular. However, the size of the output shrinks at each layer. If the input image is of size $m \times m$ and the kernel is of size $k \times k$, the output will be of size $m - k + 1 \times m - k + 1$. The rate of this shrinkage can be dramatic if the kernels used are large. Since the shrinkage is greater than 0, it limits the number of convolutional layers that can be included in the network. As layers are added, the spatial dimension of the network will eventually drop to $1 \times 1$, at which point additional layers cannot meaningfully be considered convolutional. Another special case of the zero-padding setting is when just enough zero-padding is added to keep the size of the output equal to the size of the input. MATLAB calls this *same* convolution. In this case, the network can contain as many convolutional layers as the available hardware can support, since the operation of convolution does not modify the architectural possibilities available to the next layer. However, the input pixels near the border influence fewer output pixels than the input pixels near the center. This can make the border pixels somewhat underrepresented in the model. This motivates the other extreme case, which MATLAB refers to as *full convolution*, in which enough zeroes are added for every pixel to be visited $k$ times in each direction, resulting in an output image of size $m + k - 1 \times m + k - 1$. In this case, the output pixels near the border are a function of fewer pixels than the output pixels near the center. This can make it difficult to learn a single kernel that performs well at all positions in the convolutional feature map. Usually the optimal amount of zero padding (in terms of test set classification accuracy) lies somewhere between "valid" and "same" convolution.

In some cases, we do not actually want to use convolution, but rather locally connected layers. In this case, the adjacency matrix in the graph of our MLP is the same, but every connection has its own weight, specified by a 6-D tensor **W**.
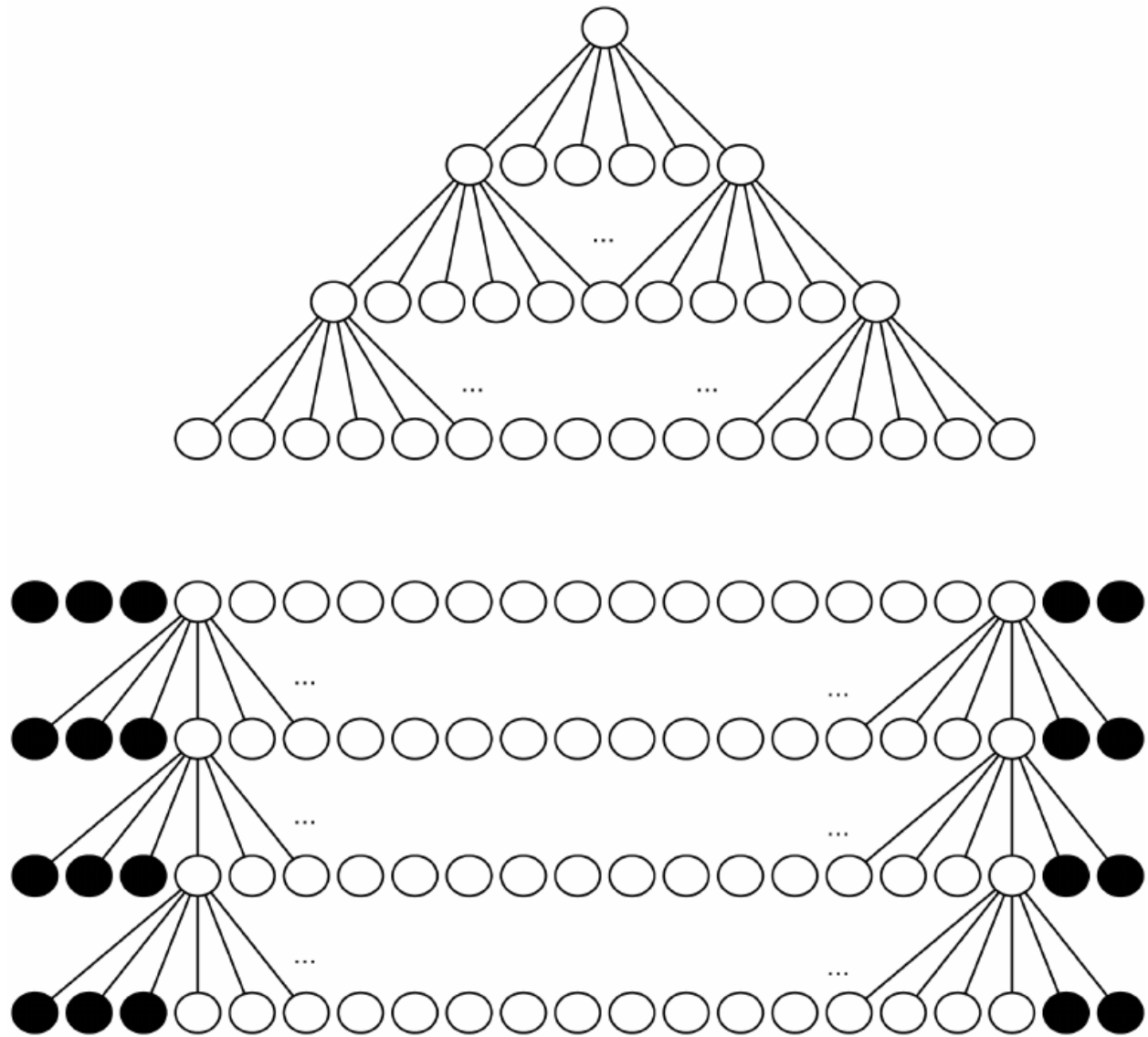
Figure 9.11: *The effect of zero padding on network size*: Consider a convolutional network with a kernel of width six at every layer. In this example, do not use any pooling, so only the convolution operation itself shrinks the network size. *Top)* In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive and some shrinking is inevitable in this kind of architecture. *Bottom)* By adding five implicit zeroes to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network.

Figure 9.12: TODO

The indices into **W** are respectively: $i$, the output channel, $j$, the output row, $k$, the output column, $l$, the input channel, $m$, the row offset within the input, and $n$, the column offset within the input. The linear part of a locally connected layer is then given by

$$Z_{i,j,k} = \sum_{l,m,n} \left[ V_{l,j+m,k+n} w_{i,j,k,l,m,n} \right].$$

This is sometimes also called *unshared convolution*, because it is a similar operation to discrete convolution with a small kernel, but without sharing parameters across locations.

TODO: Mehdi asks for a local convolution figure, showing layers in 1D topology and comparing it to fully connected layer

Locally connected layers are useful when we know that each feature should be a function of a small part of space, but there is no reason to think that the same feature should occur across all of space. For example, if we want to tell if an image is a picture of a face, we only need to look for the mouth in the bottom half of the image.

It can also be useful to make versions of convolution or locally connected layers in which the connectivity is further restricted, for example to constraint that each output channel $i$ be a function of only a subset of the input channels $l$. TODO: explain more, this paragraph just kind of dies. include a figure

*Tiled convolution* (Gregor and LeCun, 2010; Le *et al.*, 2010) offers a compromise between a convolutional layer and a locally connected layer. Rather than learning a separate set of weights at *every* spatial location, we learn a set of kernels that we rotate through as we move through space. This means that immediately neighboring locations will have different filters, like in a locally connected layer, but the memory requirements for storing the parameters will increase only by a factor of the size of this set of kernels, rather than the size of the entire output feature map. See Fig. 9.12 for a graphical depiction of tiled convolution.

To define tiled convolution algebraically, let $k$ be a 6-D tensor, where two of the dimensions correspond to different locations in the output map. Rather than having a separate index for each location in the output map, output locations cycle through a set of $t$ different choices of kernel stack in each direction. If $t$ is equal to the output width, this is the same as a locally connected layer.

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m,k+n} K_{i,l,m,n,j\%t,k\%t}$$

It is straightforward to generalize this equation to use a different tiling range

for each dimension.

Both locally connected layers and tiled convolutional layers have an interesting interaction with max-pooling: the detector units of these layers are driven by different filters. If these filters learn to detect different transformed versions of the same underlying features, then the max-pooled units become invariant to the learned transformation (see Fig. 9.9). Convolutional layers are hard-coded to be invariant specifically to translation.

Other operations besides convolution are usually necessary to implement a convolutional network. To perform learning, one must be able to compute the gradient with respect to the kernel, given the gradient with respect to the outputs. In some simple cases, this operation can be performed using the convolution operation, but many cases of interest, including the case of stride greater than 1, do not have this property.

Recall that convolution is a linear operation and can thus be described as a matrix multiplication (if we first reshape the input tensor into a flat vector). The matrix involved is a function of the convolution kernel. The matrix is sparse and each element of the kernel is copied to several elements of the matrix. It is not usually practical to implement convolution in this way, but this view helps us to derive some of the other operations needed to implement a convolutional network.

Multiplication by the transpose of the matrix defined by convolution is one such operation. This is the operation needed to backpropagate error derivatives through a convolutional layer, so it is needed to train convolutional networks that have more than one hidden layer. This same operation is also needed to compute the reconstruction in a convolutional autoencoder (or to perform the analogous role in a convolutional RBM, sparse coding model, etc.). Like the kernel gradient operation, this input gradient operation can be implemented using a convolution in some cases, but in the general case requires a third operation to be implemented. Care must be taken to coordinate this transpose operation with the forward propagation. The size of the output that the transpose operation should return depends on the zero padding policy and stride of the forward propagation operation, as well as the size of the forward propagation's output map. In some cases, multiple sizes of input to forward propagation can result in the same size of output map, so the transpose operation must be explicitly told what the size of the original input was.

It turns out that these three operations–convolution, backprop from output to weights, and backprop from output to inputs–are sufficient to compute all of the gradients needed to train any depth of feedforward convolutional network, as well as to train convolutional networks with reconstruction functions based on the transpose of convolution. See (Goodfellow, 2010) for a full derivation of the equations in the fully general multi-dimensional, multi-example case. To

give a sense of how these equations work, we present the two dimensional, single example version here.

Suppose we want to train a convolutional network that incorporates strided convolution of kernel stack $\mathbf{K}$ applied to multi-channel image $\mathbf{V}$ with stride $s$ is defined by $c(\mathbf{K}, \mathbf{V}, s)$ as in equation 9.1. Suppose we want to minimize some loss function $J(\mathbf{V}, \mathbf{K})$. During forward propagation, we will need to use $c$ itself to output $\mathbf{Z}$, which is then propagated through the rest of the network and used to compute $J$. . During backpropagation, we will receive a tensor $\mathbf{G}$ such that $G_{i,j,k} = \frac{\partial}{\partial Z_{i,j,k}} J(\mathbf{V}, \mathbf{K})$.

To train the network, we need to compute the derivatives with respect to the weights in the kernel. To do so, we can use a function

$$ g(\mathbf{G}, \mathbf{V}, s)_{i,j,k,l} = \frac{\partial}{\partial K_{i,j,k,l}} J(\mathbf{V}, \mathbf{K}) = \sum_{m,n} G_{i,m,n} V_{j, m \times s + k, n \times s + l}. $$

If this layer is not the bottom layer of the network, we'll need to compute the gradient with respect to $\mathbf{V}$ in order to backpropagate the error farther down. To do so, we can use a function

$$ h(\mathbf{K}, \mathbf{G}, s)_{i,j,k} = \frac{\partial}{\partial V_{i,j,k}} J(\mathbf{V}, \mathbf{K}) = \sum_{l,m|s \times l + m = j} \sum_{n,p|s \times n + p = k} \sum_{q} K_{q,i,m,p} G_{i,l,n}. $$

We could also use $h$ to define the reconstruction of a convolutional autoencoder, or the probability distribution over visible given hidden units in a convolutional RBM or sparse coding model. Suppose we have hidden units $\mathbf{H}$ in the same format as $\mathbf{Z}$ and we define a reconstruction

$$ \mathbf{R} = h(\mathbf{K}, \mathbf{H}, s). $$

In order to train the autoencoder, we will receive the gradient with respect to $\mathbf{R}$ as a tensor $\mathbf{E}$. To train the decoder, we need to obtain the gradient with respect to $\mathbf{K}$. This is given by $g(\mathbf{H}, \mathbf{E}, s)$. To train the encoder, we need to obtain the gradient with respect to $\mathbf{H}$. This is given by $c(\mathbf{K}, \mathbf{E}, s)$. It is also possible to differentiate through $g$ using $c$ and $h$, but these operations are not needed for the backpropagation algorithm on any standard network architectures.

Generally, we do not use only a linear operation in order to transform from the inputs to the outputs in a convolutional layer. We generally also add some bias term to each output before applying the nonlinearity. This raises the question of how to share parameters among the biases. For locally connected layers it is natural to give each unit its own bias, and for tiled convolution, it is natural to share the biases with the same tiling pattern as the kernels. For convolutional layers, it is typical to have one bias per channel of the output and share it across

all locations within each convolution map. However, if the input is of known, fixed size, it is also possible to learn a separate bias at each location of the output map. Separating the biases may slightly reduce the statistical efficiency of the model, but also allows the model to correct for differences in the image statistics at different locations. For example, when using implicit zero padding, detector units at the edge of the image receive less total input and may need larger biases.

## 9.6 Structured Outputs

TODO show diagram of an exclusively convolutional net, like for image inpainting or segmentation (Farabet? Collobert?)

## 9.7 Convolutional Modules

TODO history of ReLU -¿ maxout -¿ NIN -¿ inception

## 9.8 Data Types

The data used with a convolutional network usually consists of several channels, each channel being the observation of a different quantity at some point in space or time. See Table 9.1 for examples of data types with different dimensionalities and number of channels.

So far we have discussed only the case where every example in the train and test data has the same spatial dimensions. One advantage to convolutional networks is that they can also process inputs with varying spatial extents. These kinds of input simply cannot be represented by traditional, matrix multiplication-based neural networks. This provides a compelling reason to use convolutional networks even when computational cost and overfitting are not significant issues.

For example, consider a collection of images, where each image has a different width and height. It is unclear how to apply matrix multiplication. Convolution is straightforward to apply; the kernel is simply applied a different number of times depending on the size of the input, and the output of the convolution operation scales accordingly. Sometimes the output of the network is allowed to have variable size as well as the input, for example if we want to assign a class label to each pixel of the input. In this case, no further design work is necessary. In other cases, the network must produce some fixed-size output, for example if we want to assign a single class label to the entire image. In this case we must make some additional design steps, like inserting a pooling layer whose pooling regions scale in size proportional to the size of the input, in order to maintain a fixed number of pooled outputs.

|  | Single channel | Multi-channel |
|---|---|---|
| 1-D | Audio waveform: The axis we convolve over corresponds to time. We discretize time and measure the amplitude of the waveform once per time step. | Skeleton animation data: Animations of 3-D computer-rendered characters are generated by altering the pose of a "skeleton" over time. At each point in time, the pose of the character is described by a specification of the angles of each of the joints in the character's skeleton. Each channel in the data we feed to the convolutional model represents the angle about one axis of one joint. |
| 2-D | Audio data that has been pre-processed with a Fourier transform: We can transform the audio waveform into a 2D tensor with different rows corresponding to different frequencies and different columns corresponding to different points in time. Using convolution in the time makes the model equivariant to shifts in time. Using convolution across the frequency axis makes the model equivariant to frequency, so that the same melody played in a different octave produces the same representation but at a different height in the network's output. | Color image data: One channel contains the red pixels, one the green pixels, and one the blue pixels. The convolution kernel moves over both the horizontal and vertical axes of the image, conferring translation equivariance in both directions. |
| 3-D | Volumetric data: A common source of this kind of data is medical imaging technology, such as CT scans. | Color video data: One axis corresponds to time, one to the height of the video frame, and one to the width of the video frame. |

Table 9.1: Examples of different formats of data that can be used with convolutional networks.

Note that the use of convolution for processing variable sized inputs only makes sense for inputs that have variable size because they contain varying amounts of observation of the same kind of thing–different lengths of recordings over time, different widths of observations over space, etc. Convolution does not make sense if the input has variable size because it can optionally include different kinds of observations. For example, if we are processing college applications, and our features consist of both grades and standardized test scores, but not every applicant took the standardized test, then it does not make sense to convolve the same weights over both the features corresponding to the grades and the features corresponding to the test scores.

## 9.9 Efficient Convolution Algorithms

Modern convolutional network applications often involve networks containing more than one million units. Powerful implementations exploiting parallel computation resources, as discussed in Chapter 12.1 are essential. However, in many cases it is also possible to speed up convolution by selecting an appropriate convolution algorithm.

Convolution is equivalent to converting both the input and the kernel to the frequency domain using a Fourier transform, performing point-wise multiplication of the two signals, and converting back to the time domain using an inverse Fourier transform. For some problem sizes, this can be faster than the naive implementation of discrete convolution.

When a $d$-dimensional kernel can be expressed as the outer product of $d$ vectors, one vector per dimension, the kernel is called *separable*. When the kernel is separable, naive convolution is inefficient. It is equivalent to compose $d$ one-dimensional convolutions with each of these vectors. The composed approach is significantly faster than performing one $k$-dimensional convolution with their outer product. The kernel also takes fewer parameters to represent as vectors. If the kernel is $w$ elements wide in each dimension, then naive multidimensional convolution requires $O(w^d)$ runtime and parameter storage space, while separable convolution requires $O(w \times d)$ runtime and parameter storage space. Of course, not every convolution can be represented in this way.

Devising faster ways of performing convolution or approximate convolution without harming the accuracy of the model is an active area of research. Even techniques that improve the efficiency of only forward propagation are useful because in the commercial setting, it is typical to devote many more resources to deployment of a network than to its training.

## 9.10 Random or Unsupervised Features

Typically, the most expensive part of convolutional network training is learning the features. The output layer is usually relatively inexpensive due to the small number of features provided as input to this layer after passing through several layers of pooling. When performing supervised training with gradient descent, every gradient step requires a complete run of forward propagation and backward propagation through the entire network. One way to reduce the cost of convolutional network training is to use features that are not trained in a supervised fashion.

There are two basic strategies for obtaining convolution kernels without supervised training. One is to simply initialize them randomly. The other is to learn them with an unsupervised criterion. This approach allows the features to be determined separately from the classifier layer at the top of the architecture. One can then extract the features for the entire training set just once, essentially constructing a new training set for the last layer. Learning the last layer is then typically a convex optimization problem, assuming the last layer is something like logistic regression or an SVM.

Random filters often work surprisingly well in convolutional networks (Jarrett *et al.*, 2009b; Saxe *et al.*, 2011; Pinto *et al.*, 2011; Cox and Pinto, 2011). Saxe *et al.* (2011) showed that layers consisting of convolution following by pooling naturally become frequency selective and translation invariant when assigned random weights. They argue that this provides an inexpensive way to choose the architecture of a convolutional network: first evaluate the performance of several convolutional network architectures by training only the last layer, then take the best of these architectures and train the entire architecture using a more expensive approach.

An intermediate approach is to learn the features, but using methods that do not require full forward and back-propagation at every gradient step. As with multilayer perceptrons, we use greedy layer-wise unsupervised pretraining, to train the first layer in isolation, then extract all features from the first layer only once, then train the second layer in isolation given those features, and so on. The canonical example of this is the convolutional deep belief network (Lee *et al.*, 2009). Convolutional networks offer us the opportunity to take this strategy one step further than is possible with multilayer perceptrons. Instead of training an entire convolutional layer at a time, we can actually train a small but densely-connected unsupervised model (such as PSD, described in Chapter 15.8.2, or $k$-means) of a single image *patch*. We can then use the weight matrices from this patch-based model to define the kernels of a convolutional layer. This means that it is possible to use unsupervised learning to train a convolutional network *without ever using convolution during the training process*. Using this approach, we can

train very large models and incur a high computational cost only at inference time (Ranzato *et al.*, 2007b; Jarrett *et al.*, 2009b; Kavukcuoglu *et al.*, 2010a; Coates *et al.*, 2013).

As with other approaches to unsupervised pretraining, it remains difficult to tease apart the cause of some of the benefits seen with this approach. Unsupervised pretraining may offer some regularization relative to supervised training, or it may simply allow us to train much larger architectures due to the reduced computational cost of the learning rule.

## 9.11 The Neuroscientific Basis for Convolutional Networks

Convolutional networks are perhaps the greatest success story of biologically inspired artificial intelligence. Though convolutional networks have been guided by many other fields, some of the key design principles of neural networks were drawn from neuroscience.

The history of convolutional networks begins with neuroscientific experiments long before the relevant computational models were developed. Neurophysiologists David Hubel and Torsten Wiesel collaborated for several years to determine many of the most basic facts about how the mammalian vision system works (Hubel and Wiesel, 1959, 1962, 1968). Their accomplishements were eventually recognized with a Nobel Prize. Their findings that have had the greatest influence on contemporary deep learning models were based on recording the activity of individual neurons in cats. By anesthetizing the cat, they could immobilize the cat's eye and observe how neurons in the cat's brain responded to images projected in precise locations on a screen in front of the cat.

Their worked helped to characterize many aspects of brain function that are beyond the scope of this book. From the point of view of deep learning, we can focus on a simplified, cartoon view of brain function.

In this simplified view, we focus on a part of the brain called *V1*, also known as the *primary visual cortex*. V1 is the first area of the brain that begins to perform significantly advanced processing of visual input. In this cartoon view, images are formed by light arriving in the eye and stimulating the retina, the light-sensitive tissue in the back of the eye. The neurons in the retina perform some simple preprocessing of the image but do not substantially alter the way it is represented. The image then passes through the optic nerve and a brain region called the lateral geniculate nucleus. The main role, as far as we are concerned here, of both of these anatomical regions is primarily just to carry the signal from the eye to V1, which is located at the back of the head.

A convolutional network layer is designed to capture three properties of V1:

1. V1 is arranged in a spatial map. It actually has a two-dimensional structure mirroring the structure of the image in the retina. For example, light arriving at the lower half of the retina affects only the corresponding half of V1. Convolutional networks capture this property by having their features defined in terms of two dimensional maps.

2. V1 contains many *simple cells*. A simple cell's activity can to some extent be characterized by a linear function of the image in a small, spatially localized receptive field. The detector units of a convolutional network are designed to emulate these properties of simple cells. V1 also contains many *complex cells*. These cells respond to features that are similar to those detected by simple cells, but complex cells are invariant to small shifts in the position of the feature. This inspires the pooling units of convolutional networks. Complex cells are also invariant to some changes in lighting that cannot be captured simply by pooling over spatial locations. These invariances have inspired some of the cross-channel pooling strategies in convolutional networks, such as maxout units (Goodfellow *et al.*, 2013a).

Though we know the most about V1, it is generally believed that the same basic principles apply to other brain regions. In our cartoon view of the visual system, the basic strategy of detection followed by pooling is repeatedly applied as we move deeper into the brain. As we pass through multiple anatomical layers of the brain, we eventually find cells that respond to some specific concept and are invariant to many transformations of the input. These cells have been nicknamed "grandmother cells" — the idea is that a person could have a neuron that activates when seeing an image of their grandmother, regardless of whether she appears in the left or right side of the image, whether the image is a close-up of her face or zoomed out shot of her entire body, whether she is brightly lit, or in shadow, etc.

These grandmother cells have been shown to actually exist in the human brain, in a region called the medial temporal lobe (Quiroga *et al.*, 2005). Researchers tested whether individual neurons would respond to photos of famous individuals, and found what has come to be called the "Halle Berry neuron": an individual neuron that is activated by the concept of Halle Berry. This neuron fires when a person sees a photo of Halle Berry, a drawing of Halle Berry, or even text containing the words "Halle Berry." Of course, this has nothing to do with Halle Berry herself; other neurons responded to the presence of Bill Clinton, Jennifer Aniston, etc.

These medial temporal lobe neurons are somewhat more general than modern convolutional networks, which would not automatically generalize to identifying a person or object when reading its name. The closest analog to a convolutional network's last layer of features is a brain area called the inferotemporal cortex (IT). When viewing an object, information flows from the retina, through the

LGN, to V1, then onward to V2, then V4, then IT. This happens within the first 100ms of glimpsing an object. If a person is allowed to continue looking at the object for more time, then information will begin to flow backwards as the brain uses top-down feedback to update the activations in the lower level brain areas. However, if we interrupt the person's gaze, and observe only the firing rates that result from the first 100ms of mostly feed-forward activation, then IT proves to be very similar to a convolutional network. Convolutional networks can predict IT firing rates, and also perform very similarly to (time limited) humans on object recognition tasks (DiCarlo, 2013).

That being said, there are many differences between convolutional networks and the mammalian vision system. Some of these differences are well known to computational neuroscientists, but outside the scope of this book. Some of these differences are not yet known, because many basic questions about how the mamalian vision system works. As a brief list:

- The human eye is mostly very low resolution, except for a tiny patch called the *fovea*. The fovea only observes an area about the size of a thumbnail held at arms length. Though we feel as if we can see an entire scene in high resolution, this is an illusion created by the subconscious part of our brain, as it stitches together several glimpses of small areas. Most convolutional networks actual receive large full resolution photographs as input.

- The human visual system is integrated with many other senses, such as hearing, and factors like our moods and thoughts. Convolutional networks so far are purely visual.

- The human visual system does much more than just recognize objects. It is able to understand entire scenes including many objects and relationships between objects, and processes rich 3-D geometric information needed for our bodies to interface with the world. Convolutional networks have been applied to some of these problems but these applications are in their infancy.

- Even simple brain areas like V1 are heavily impacted by feedback from higher levels. Feedback has been explored extensively in neural network models but has not yet been shown to offer a compelling improvement.

- While feed-forward IT firing rates capture much of the same information as convolutional network features, it's not clear how similar the intermediate computations are. The brain probably uses very different activation and pooling functions. An individual neuron's activation probably is not well-characterized by a single linear filter response. A recent model of V1 involves multiple quadratic filters for each neuron (Rust *et al.*, 2005). Indeed our

cartoon picture of "simple cells" and "complex cells" might create a non-existent distinction; simple cells and complex cells might both be the same kind of cell but with their "parameters" enabling a continuum of behaviors ranging from what we call "simple" to what we call "complex."

It's also worth mentioning that neuroscience has told us relatively little about how to *train* convolutional networks. Model structures with parameter sharing across multiple spatial locations date back to early connectionist models of vision (Marr and Poggio, 1976), but these models did not use the modern backpropagation algorithm and gradient descent. For example, the Neocognitron (Fukushima, 1980) incorporated most of the model architecture design elements of the modern convolutional network but relied on a layerwise unsupervised clustering algorithm.

Lang and Hinton (1988) introduced the use of backpropagation to train *time-delay neural networks* (TDNNs). To use contemporary terminology, TDNNs are one-dimensional convolutional networks applied to time series. Back-propagation applied to these models was not inspired by any neuroscientific observation and is considered by some to be biologically implausible. Following the success of backpropagation-based training of TDNNs, (LeCun *et al.*, 1989) developed the modern convolutional network by applying the same training algorithm to 2-D convolution applied to images.

So far we have described how simple cells are roughly linear and selective for certain features, complex cells are more non-linear and become invariant to some transformations of these simple cell features, and stacks of layers that alternative between selectivity and invariance can yield grandmother cells for very specific phenomena. We have not yet described precisely what these individual cells detect. In a deep, nonlinear network, it can be difficult to understand the function of individual cells. Simple cells in the first layer are easier to analyze, because their responses are driven by a linear function. In an artificial neural network, we can just display an image of the kernel to see what the corresponding channel of a convolutional layer responds to. In a biological neural network, we do not have access to the weights themselves. Instead, we put an electrode in the neuron itself, display several samples of white noise images in front of the animal's retina, and record how each of these samples causes the neuron to activate. We can then fit a linear model to these responses in order to obtain an approximation of the neuron's weights. This approach is known as *reverse correlation* (Ringach and Shapley, 2004).

Reverse correlation shows us that most V1 cells have weights that are described by *Gabor functions*. The Gabor function describes the weight at a 2-D point in the image. We can think of an image as being a function of 2-D coordinates, $I(x, y)$. Likewise, we can think of a simple cell as sampling the image at a set of

locations, defined by a set of $x$ coordinates $\mathbb{X}$ and a set of $y$ coordinates, $\mathbb{Y}$, and applying weights that are also a function of the location, $w(x, y)$. From this point of view, the response of a simple cell to an image is given by

$$s(I) = \sum_{x \in \mathbb{X}} \sum_{y \in \mathbb{Y}} w(x, y) I(x, y).$$

Specifically, $w(x, y)$ takes the form of a Gabor function:

$$w(x, y; \alpha, \beta_x, \beta_y, f, \phi, x_0, y_0, \tau) = \alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right) \cos(f x' + \phi),$$

where

$$x' = (x - x_0) \cos(\tau) + (y - y_0) \sin(\tau)$$

and

$$y' = -(x - y_0) \sin(\tau) + (y - y_0) \cos(\tau).$$

Here, $\alpha$, $\beta_x$, $\beta_y$, $f$, $\phi$, $x_0$, $y_0$, and $\tau$ are parameters that control the properties of the Gabor function. Fig. 9.13 shows some examples of Gabor functions with different settings of these parameters.

The parameters $x_0$, $y_0$, and $\tau$ define a coordinate system. We translate and rotate $x$ and $y$ to form $x'$ and $y'$. Specifically, the simple cell will respond to image features centered at the point $(x_0, y_0)$, and it will respond to changes in brightness as we move along a line rotated $\tau$ radians from the horizontal.

Viewed as a function of $x'$ and $y'$, the function $w$ then responds to changes in brightness as we move along the $x'$ axis. It has two important factors: one is a Gaussian function and the other is a cosine function.

The Gaussian factor $\alpha \exp\left(-\beta_x x'^2 - \beta_y y'^2\right)$ can be seen as a gating term that ensures the simple cell will only respond to values near where $x'$ and $y'$ are both zero, in other words, near the center of the cell's receptive field. The scaling factor $\alpha$ adjusts the total magnitude of the simple cell's response, while $\beta_x$ and $\beta_y$ control how quickly its receptive field falls off.

The cosine factor $\cos(f x' + \phi)$ controls how the simple cell responds to changing brightness along the $x'$ axis. The parameter $f$ controls the frequency of the cosine and $\phi$ controls its phase offset.

Altogether, this cartoon view of simple cells means that a simple cell responds to a specific spatial frequency of brightness in a specific direction at a specific location. They are most excited when the wave of brightness in the image has the same phase as the weights (i.e., when the image is bright where the weights are positive and dark where the weights are negative) and are most inhibited when the wave of brightness is fully out of phase with the weights (i.e., when the image is dark where the weights are positive and bright where the weights are negative).
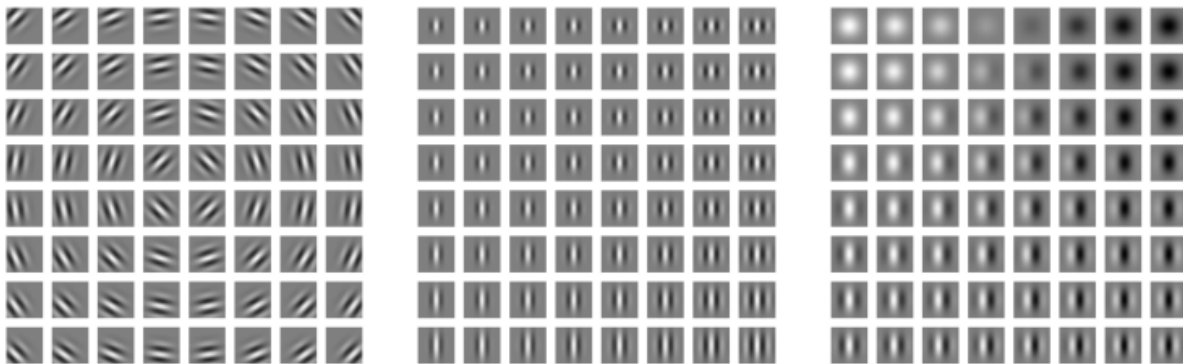
Figure 9.13: Gabor functions with a variety of parameter settings. White indicates large positive weight, black indicates large negative weight, and the background gray corresponds to zero weight. *Left)* Gabor functions with different values of the parameters that control the coordinate system: $x_0$, $y_0$, and $\tau$. Each gabor function in this grid is assigned a value of $x_0$ and $y_0$ proportional to its position in its grid, and $\tau$ is chosen so that each Gabor is sensitive to the direction radiating out from the center of the grid. For the other two plots, $x_0$, $y_0$, and $\tau$ are fixed to zero. *Center)* Gabor functions with different Gaussian scale parameters $beta_x$ and $\beta_y$. Gabor functions are arranged in increasing width (decreasing $\beta_x$) as we move left to right through the grid, and increasing height (decreasing $\beta_y$) as we move top to bottom. For the other two plots, the $\beta$ values are fixed to $1.5\times$ the image width. *Right)* Gabor functions with different sinusoid parameters $f$ and $\phi$. As we move top to bottom, $f$ increases, and as we move left to right, $\phi$ increases. For the other two plots, $\phi$ is fixed to 0 and $f$ is fixed to $5\times$ the image width.

The cartoon view of a complex cell is that it computes the $L^2$ norm of the 2-D vector containing two simple cell's responses: $c(I) = \sqrt{s_0(I)^2 + s_1(I)^2}$. An important special case occurs when $s_1$ has all of the same parameters as $s_0$ except for $\phi$, and $\phi$ is set such that $s_1$ is one quarter cycle out of phase with $s_0$. In this case, $s_0$ and $s_1$ form a *quadrature pair*. A complex cell defined in this way responds when the Gaussian reweighted image $I(x, y) \exp(-\beta_x x'^2 - \beta_y y'^2)$ contains a high amplitude sinusoidal wave with frequency $f$ in direction $\tau$ near $(x_0, y_0)$, *regardless of the phase offset of this wave.* In other words, the complex cell is invariant to small translations of the image in direction $\tau$, or to negating the image (replacing black with white and vice versa).

Some of the most striking correspondences between neuroscience and machine learning come from visually comparing the features learned by machine learning models with those employed by V1. Olshausen and Field (1996) showed that a simple unsupervised learning algorithm, sparse coding, learns features with receptive fields similar to those of simple cells. Since then, we have found that an extremely wide variety of statistical learning algorithms learn features with Gabor-like functions when applied to natural images. This includes most deep learning algorithms, which learn these features in their first layer. Fig. 9.14 shows some examples. Because so many different learning algorithms learn edge detectors, it is difficult to conclude that any specific learning algorithm is the "right" model of the brain just based on the features that it learns (though it can certainly be a bad sign if an algorithm does *not* learn some sort of edge detector when applied to natural images). These features are an important part of the statistical structure of natural images and can be recovered by many different approaches to statistical modeling. See Hyvärinen *et al.* (2009) for a review of the field of natural image statistics.

## 9.12 Convolutional Networks and the History of Deep Learning

Convolutional networks have played an important role in the history of deep learning. They are a key example of a successful application of insights obtained by studying the brain to machine learning applications. They were also some of the first deep models to perform well, long before arbitrary deep models were considered viable. Convolutional networks were also some of the first neural networks to solve important commercial applications and remain at the forefront of commercial applications of deep learning today. For example, in the 1990's, the neural network research group at AT&T developed a convolutional network for reading checks (LeCun *et al.*, 1998d). By the end of the 1990's, this system deployed by NEC was reading over 10% of all the checks in the US. A little bit later, several
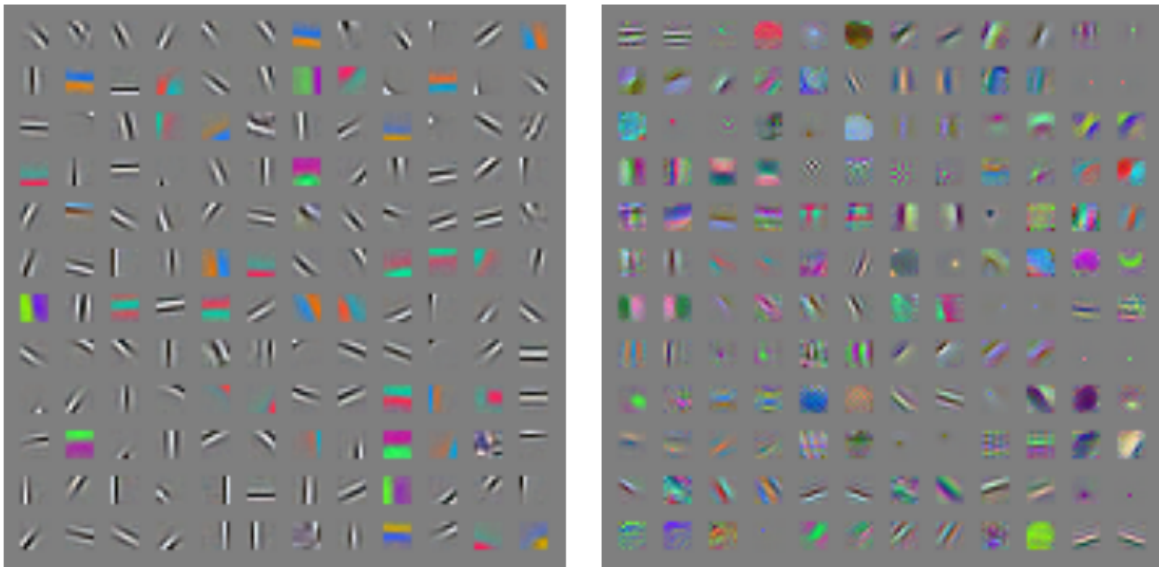
Figure 9.14: Many machine learning algorithms learn features that detect edges or specific colors of edges when applied to natural images. These feature detectors are reminiscent of the Gabor functions known to be present in primary visual cortex. *Left) Weights learned by an unsupervised learning algorithm (spike and slab sparse coding) applied to small image patches. Right)* Convolution kernels learned by the first layer of a fully supervised convolutional maxout network. Neighboring pairs of filters drive the same maxout unit.

OCR and handwriting recognitions systems based on convolutional nets were deployed by Microsoft (Simard *et al.*, 2003). See Chapter 12 for more details on such applications and more modern applications of convolutional networks. See LeCun *et al.* (2010) for a more in-depth history of convolutional networks up to 2010.

Convolutional networks were also used to win many contests. The current intensity of commercial interest in deep learning began when Krizhevsky *et al.* (2012a) won the ImageNet object recognition challenge, but convolutional networks had been used to win other machine learning and computer vision contests with less impact for years earlier.

Convolutional nets were some of first working deep networks trained with back-propagation. It is not entirely clear why convolutional networks succeeded when general backpropagation networks were considered to have failed. It may simply be that convolutional networks were more computationally efficient than fully connected networks, so it was easier to run multiple experiments with them and tune their implementation and hyperparameters. Larger networks also seem to be easier to train. With modern hardware, fully connected networks appear to perform reasonably on many tasks, even when using datasets that were available and activation functions that were popular during the times when fully connected

networks were believed not to work well. It may be that the primary barriers to the success of neural networks were psychological, e.g., having trained for much longer (rather than concluding that training had stalled out of discouragement) might have yielded much better results. Whatever the case, it is fortunate that convolutional networks performed well decades ago. In many ways, they "carried the torch" for the rest of deep learning and paved the way to the acceptance of neural networks in general.