Chapter 8

Convolutional Neural Networks

"The soul never thinks without a picture."—Aristotle

8.1 Introduction

Convolutional neural networks are designed to work with grid-structured inputs, which have strong spatial dependencies in local regions of the grid. The most obvious example of grid-structured data is a 2-dimensional image. This type of data also exhibits spatial dependencies, because adjacent spatial locations in an image often have similar color values of the individual pixels. An additional dimension captures the different colors, which creates a 3-dimensional input *volume*. Therefore, the features in a convolutional neural network have dependencies among one another based on spatial distances. Other forms of sequential data like text, time-series, and sequences can also be considered special cases of grid-structured data with various types of relationships among adjacent items. The vast majority of applications of convolutional neural networks focus on image data, although one can also use these networks for all types of temporal, spatial, and spatiotemporal data.

An important property of image data is that it exhibits a certain level of *translation invariance*, which is not the case in many other types of grid-structured data. For example, a banana has the same interpretation, whether it is at the top or the bottom of an image. Convolutional neural networks tend to create similar feature values from local regions with similar patterns. One advantage of image data is that the effects of specific inputs on the feature representations can often be described in an intuitive way. Therefore, this chapter will primarily work with the image data setting. A brief discussion will also be devoted to the applications of convolutional neural networks to other settings.

C. C. Aggarwal, *Neural Networks and Deep Learning*,
https://doi.org/10.1007/978-3-319-94463-0_8

An important defining characteristic of convolutional neural networks is an operation, which is referred to as *convolution*. A convolution operation is a dot-product operation between a grid-structured set of weights and similar grid-structured inputs drawn from different spatial localities in the input volume. This type of operation is useful for data with a high level of spatial or other locality, such as image data. Therefore, convolutional neural networks are defined as networks that use the convolutional operation in at least one layer, although most convolutional neural networks use this operation in multiple layers.

8.1.1 Historical Perspective and Biological Inspiration

Convolutional neural networks were one of the first success stories of deep learning, well before recent advancements in training techniques led to improved performance in other types of architectures. In fact, the eye-catching successes of some convolutional neural network architectures in image-classification contests after 2011 led to broader attention to the field of deep learning. Long-standing benchmarks like *ImageNet* [581] with a top-5 classification error-rate of more than 25% were brought down to less than 4% in the years between 2011 and 2015. Convolutional neural networks are well suited to the process of hierarchical feature engineering with depth; this is reflected in the fact that the deepest neural networks in all domains are drawn from the field of convolutional networks. Furthermore, these networks also represent excellent examples of how biologically inspired neural networks can sometimes provide ground-breaking results. The best convolutional neural networks today reach or exceed human-level performance, a feat considered impossible by most experts in computer vision only a couple of decades back.

The early motivation for convolutional neural networks was derived from experiments by Hubel and Wiesel on a cat's visual cortex [212]. The visual cortex has small regions of cells that are sensitive to specific regions in the visual field. In other words, if specific areas of the visual field are excited, then those cells in the visual cortex will be activated as well. Furthermore, the excited cells also depend on the shape and orientation of the objects in the visual field. For example, vertical edges cause some neuronal cells to be excited, whereas horizontal edges cause other neuronal cells to be excited. The cells are connected using a layered architecture, and this discovery led to the conjecture that mammals use these different layers to construct portions of images at different levels of abstraction. From a machine learning point of view, this principle is similar to that of hierarchical feature extraction. As we will see later, convolutional neural networks achieve something similar by encoding primitive shapes in earlier layers, and more complex shapes in later layers.

Based on these biological inspirations, the earliest neural model was the *neocognitron* [127]. However, there were several differences between this model and the modern convolutional neural network. The most prominent of these differences was that the notion of weight sharing was not used. Based on this architecture, one of the first fully convolutional architectures, referred to as *LeNet-5* [279], was developed. This network was used by banks to identify hand-written numbers on checks. Since then, the convolutional neural network has not evolved much; the main difference is in terms of using more layers and stable activation functions like the ReLU. Furthermore, numerous training tricks and powerful hardware options are available to achieve better success in training when working with deep networks and large data sets.

A factor that has played an important role in increasing the prominence of convolutional neural networks has been the annual *ImageNet* competition [582] (also referred to as *"ImageNet Large Scale Visual Recognition Challenge [ILSVRC]"*). The ILSVRC competition uses the *ImageNet* data set [581], which is discussed in Section 1.8.2 of Chapter 1. Convolutional neural networks have been consistent winners of this contest since 2012. In fact, the dominance of convolutional neural networks for image classification is so well recognized today that almost all entries in recent editions of this contest have been convolutional neural networks. One of the earliest methods that achieved success in the 2012 *ImageNet* competition by a large margin was *AlexNet* [255]. Furthermore, the improvements in accuracy have been so extraordinarily large in the last few years that it has changed the landscape of research in the area. In spite of the fact that the vast majority of eye-catching performance gains have occurred from 2012 to 2015, the architectural differences between recent winners and some of the earliest convolutional neural networks are rather small at least at a conceptual level. Nevertheless, small details seem to matter a lot when working with almost all types of neural networks.

8.1.2 Broader Observations About Convolutional Neural Networks

The secret to the success of any neural architecture lies in tailoring the structure of the network with a semantic understanding of the domain at hand. Convolutional neural networks are heavily based on this principle, because they use sparse connections with a high-level of parameter-sharing in a domain-sensitive way. In other words, not all states in a particular layer are connected to those in the previous layer in an indiscriminate way. Rather, the value of a feature in a particular layer is connected only to a local spatial region in the previous layer with a consistent set of shared parameters across the full spatial footprint of the image. This type of architecture can be viewed as a domain-aware regularization, which was derived from the biological insights in Hubel and Wiesel's early work. In general, the success of the convolutional neural network has important lessons for other data domains. A carefully designed architecture, in which the relationships and dependencies among the data items are used in order to reduce the parameter footprint, provides the key to results of high accuracy.

A significant level of domain-aware regularization is also available in recurrent neural networks, which share the parameters from different temporal periods. This sharing is based on the assumption that temporal dependencies remain invariant with time. Recurrent neural networks are based on intuitive understanding of temporal relationships, whereas convolutional neural networks are based on an intuitive understanding of spatial relationships. The latter intuition was directly extracted from the organization of biological neurons in a cat's visual cortex. This outstanding success provides a motivation to explore how neuroscience may be leveraged to design neural networks in clever ways. Even though artificial neural networks are only caricatures of the true complexity of the biological brain, one should not underestimate the intuition that one can obtain by studying the basic principles of neuroscience [176].

Chapter Organization

This chapter is organized as follows. The next section will introduce the basics of a convolutional neural network, the various operations, and the way in which they are organized. The training process for convolutional networks is discussed in Section 8.3. Case studies with some typical convolutional neural networks that have won recent competitions are discussed in Section 8.4. The convolutional autoencoder is discussed in Section 8.5. A variety of applications of convolutional networks are discussed in Section 8.6. A summary is given in Section 8.7.

8.2 The Basic Structure of a Convolutional Network

In convolutional neural networks, the states in each layer are arranged according to a spatial grid structure. These spatial relationships are inherited from one layer to the next because each feature value is based on a small local spatial region in the previous layer. It is important to maintain these spatial relationships among the grid cells, because the convolution operation and the transformation to the next layer is critically dependent on these relationships. Each layer in the convolutional network is a 3-dimensional grid structure, which has a *height*, *width*, and *depth*. The depth of a layer in a convolutional neural network should not be confused with the depth of the network itself. The word "depth" (when used in the context of a single layer) refers to the number of *channels* in each layer, such as the number of primary color channels (e.g., blue, green, and red) in the input image or the number of feature maps in the hidden layers. The use of the word "depth" to refer to both the number of feature maps in each layer as well as the number of layers is an unfortunate overloading of terminology used in convolutional networks, but we will be careful while using this term, so that it is clear from its context.

The convolutional neural network functions much like a traditional feed-forward neural network, except that the operations in its layers are spatially organized with sparse (and carefully designed) connections between layers. The three types of layers that are commonly present in a convolutional neural network are *convolution*, *pooling*, and *ReLU*. The ReLU activation is no different from a traditional neural network. In addition, a final set of layers is often fully connected and maps in an application-specific way to a set of output nodes. In the following, we will describe each of the different types of operations and layers, and the typical way in which these layers are interleaved in a convolutional neural network.

Why do we need depth in each layer of a convolutional neural network? To understand this point, let us examine how the input to the convolutional neural network is organized. The input data to the convolutional neural network is organized into a 2-dimensional grid structure, and the values of the individual grid points are referred to as *pixels*. Each pixel, therefore, corresponds to a spatial location within the image. However, in order to encode the precise color of the pixel, we need a multidimensional array of values at each grid location. In the RGB color scheme, we have an intensity of the three primary colors, corresponding to red, green, and blue, respectively. Therefore, if the spatial dimensions of an image are 32×32 pixels and the depth is 3 (corresponding to the RGB color channels), then the overall number of pixels in the image is $32 \times 32 \times 3$. This particular image size is quite common, and also occurs in a popularly used data set for benchmarking, known as CIFAR-10 [583]. An example of this organization is shown in Figure 8.1(a). It is natural to represent the input layer in this 3-dimensional structure because two dimensions are devoted to spatial relationships and a third dimension is devoted to the independent properties along these channels. For example, the intensities of the primary colors are the independent properties in the first layer. In the hidden layers, these independent properties correspond to various types of shapes extracted from local regions of the image. For the purpose of discussion, assume that the input in the qth layer is of size $L_q \times B_q \times d_q$. Here, L_q refers to the *height* (or length), B_q refers to the width (or breadth), and d_q is the depth. In almost all image-centric applications, the values of L_q and B_q are the same. However, we will work with separate notations for height and width in order to retain generality in presentation.

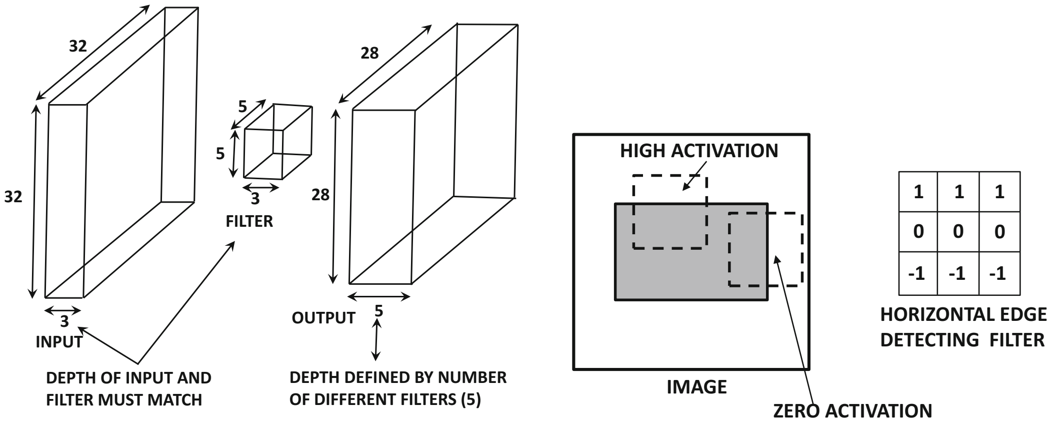
For the first (input) layer, these values are decided by the nature of the input data and its preprocessing. In the above example, the values are $L_1 = 32$, $B_1 = 32$, and $d_1 = 3$. Later layers have exactly the same 3-dimensional organization, except that each of the d_q 2-dimensional grid of values for a particular input can no longer be considered a grid of raw pixels. Furthermore, the value of d_q is much larger than three for the hidden layers because the number of independent properties of a given local region that are relevant to classification can be quite significant. For $q > 1$, these grids of values are referred to as *feature maps* or *activation maps*. These values are analogous to the values in the hidden layers in a feed-forward network.

In the convolutional neural network, the parameters are organized into sets of 3-dimensional structural units, known as *filters* or *kernels*. The filter is usually square in terms of its spatial dimensions, which are typically much smaller than those of the layer the filter is applied to. On the other hand, *the depth of a filter is always same is the same as that of the layer to which it is applied*. Assume that the dimensions of the filter in the qth layer are $F_q \times F_q \times d_q$. An example of a filter with $F_1 = 5$ and $d_1 = 3$ is shown in Figure 8.1(a). It is common for the value of F_q to be small and odd. Examples of commonly used values of F_q are 3 and 5, although there are some interesting cases in which it is possible to use $F_q = 1$.

The *convolution operation* places the filter at each possible position in the image (or hidden layer) so that the filter fully overlaps with the image, and performs a dot product between the $F_q \times F_q \times d_q$ parameters in the filter and the matching grid in the input volume (with same size $F_q \times F_q \times d_q$). The dot product is performed by treating the entries in the relevant 3-dimensional region of the input volume and the filter as vectors of size $F_q \times F_q \times d_q$, so that the elements in both vectors are ordered based on their corresponding positions in the grid-structured volume. How many possible positions are there for placing the filter? This question is important, because each such position therefore defines a spatial "pixel" (or, more accurately, a *feature*) in the next layer. In other words, the number of alignments between the filter and image defines the spatial height and width of the next hidden layer. The relative spatial positions of the features in the next layer are defined based on the relative positions of the upper left corners of the corresponding spatial grids in the previous layer. When performing convolutions in the qth layer, one can align the filter at $L_{q+1} = (L_q - F_q + 1)$ positions along the height and $B_{q+1} = (B_q - F_q + 1)$ along the width of the image (without having a portion of the filter "sticking out" from the borders of the image). This results in a total of $L_{q+1} \times B_{q+1}$ possible dot products, which defines the size of the next hidden layer. In the previous example, the values of L_2 and B_2 are therefore defined as follows:

$$L_2 = 32 - 5 + 1 = 28$$
$$B_2 = 32 - 5 + 1 = 28$$

The next hidden layer of size 28 × 28 is shown in Figure 8.1(a). However, this hidden layer also has a depth of size $d_2 = 5$. Where does this depth come from? This is achieved by using 5 different filters with their own independent sets of parameters. Each of these 5 sets of spatially arranged features obtained from the output of a single filter is referred to as a *feature map*. Clearly, an increased number of feature maps is a result of a larger number of filters (i.e., parameter footprint), which is $F_q^2 \cdot d_q \cdot d_{q+1}$ for the qth layer. *The number of filters used in each layer controls the capacity of the model because it directly controls the number of parameters.* Furthermore, increasing the number of filters in a particular layer increases the number of feature maps (i.e., depth) of the next layer. It is possible for different layers to have very different numbers of feature maps, depending on the number of filters we use for the convolution operation in the previous layer. For example, the input layer typically only has three color channels, but it is possible for the each of the later hidden layers to have depths (i.e., number of feature maps) of more than 500. The idea here is that each filter tries to identify a particular type of spatial pattern in a small rectangular region of the image, and therefore a large number of filters is required to capture a broad variety of the possible shapes that are combined to create the final image (unlike the case of the input layer, in which three RGB channels are sufficient). Typically, the later layers tend to have a smaller spatial footprint, but greater depth in terms of the number of feature maps. For example, the filter shown in Figure 8.1(b) represents a horizontal edge detector on a grayscale image with one channel. As shown in Figure 8.1(b), the resulting feature will have high activation at each position where a horizontal edge is seen. A perfectly vertical edge will give zero activation, whereas a slanted edge might give intermediate activation. Therefore, sliding the filter everywhere in the image will already detect several key outlines of the image in a single feature map of the output volume. Multiple filters are used to create an output volume with more than one feature map. For example, a different filter might create a spatial feature map of vertical edge activations.


32
32
3
INPUT
5
5
3
FILTER
28
28
OUTPUT
5
DEPTH OF INPUT AND FILTER MUST MATCH
DEPTH DEFINED BY NUMBER OF DIFFERENT FILTERS (5)
HIGH ACTIVATION
IMAGE
ZERO ACTIVATION
HORIZONTAL EDGE DETECTING FILTER

1	1	1
0	0	0
-1	-1	-1

(a) (b)

Figure 8.1: (a) The convolution between an input layer of size 32 × 32 × 3 and a filter of size 5 × 5 × 3 produces an output layer with spatial dimensions 28 × 28. The depth of the resulting output depends on the number of distinct filters and not on the dimensions of the input layer or filter. (b) Sliding a filter around the image tries to look for a particular feature in various windows of the image.

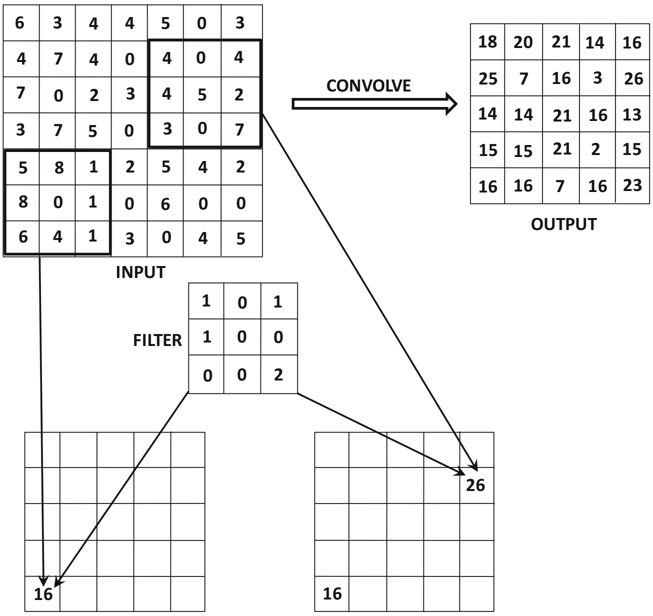
We are now ready to formally define the convolution operation. The pth filter in the qth layer has parameters denoted by the 3-dimensional tensor $W^{(p,q)} = [w_{ijk}^{(p,q)}]$. The indices i, j, k indicate the positions along the height, width, and depth of the filter. The feature maps in the qth layer are represented by the 3-dimensional tensor $H^{(q)} = [h_{ijk}^{(q)}]$. When the value of q is 1, the special case corresponding to the notation $H^{(1)}$ simply represents the input layer (which is not hidden). Then, the convolutional operations from the qth layer to the $(q+1)$th layer are defined as follows:

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \forall i \in \{1 \ldots, L_q - F_q + 1\}$$

$$\forall j \in \{1 \ldots B_q - F_q + 1\}$$

$$\forall p \in \{1 \ldots d_{q+1}\}$$

The expression above seems notationally complex, although the underlying convolutional operation is really a simple dot product over the entire volume of the filter, which is repeated over all valid spatial positions (i, j) and filters (indexed by p). It is intuitively helpful to understand a convolution operation by placing the filter at each of the 28×28 possible spatial positions in the first layer of Figure 8.1(a) and performing a dot product between the vector of $5 \times 5 \times 3 = 75$ values in the filter and the corresponding 75 values in $H^{(1)}$. Even though the size of the input layer in Figure 8.1(a) is 32×32, there are only $(32-5+1) \times (32-5+1)$ possible spatial alignments between an input volume of size 32×32 and a filter of size 5×5.



INPUT

6	3	4	4	5	0	3
4	7	4	0	4	0	4
7	0	2	3	4	5	2
3	7	5	0	3	0	7
5	8	1	2	5	4	2
8	0	1	0	6	0	0
6	4	1	3	0	4	5

FILTER

1	0	1
1	0	0
0	0	2

OUTPUT

18	20	21	14	16
25	7	16	3	26
14	14	21	16	13
15	15	21	2	15
16	16	7	16	23

Figure 8.2: An example of a convolution between a $7 \times 7 \times 1$ input and a $3 \times 3 \times 1$ filter with stride of 1. A depth of 1 has been chosen for the filter/input for simplicity. For depths larger than 1, the contributions of each input feature map will be added to create a single value in the feature map. A single filter will always create a single feature map irrespective of its depth.

The convolution operation brings to mind Hubel and Wiesel's experiments that use the activations in small regions of the visual field to activate particular neurons. In the case of convolutional neural networks, this visual field is defined by the filter, which is applied to all locations of the image in order to detect the presence of a shape at each spatial location. Furthermore, the filters in earlier layers tend to detect more primitive shapes, whereas the filters in later layers create more complex compositions of these primitive shapes. This is not particularly surprising because most deep neural networks are good at hierarchical feature engineering.

One property of convolution is that it shows *equivariance to translation.* In other words, if we shifted the pixel values in the input in any direction by one unit and then applied convolution, the corresponding feature values will shift with the input values. This is because of the shared parameters of the filter across the entire convolution. The reason for sharing parameters across the entire convolution is that the presence of a particular shape in any part of the image should be processed in the same way irrespective of its specific spatial location.

In the following, we provide an example of the convolution operation. In Figure 8.2, we have shown an example of an input layer and a filter with depth 1 for simplicity (which does occur in the case of grayscale images with a single color channel). Note that the depth of a layer must exactly match that of its filter/kernel, and the contributions of the dot products over all the feature maps in the corresponding grid region of a particular layer will need to be added (in the general case) to create a single output feature value in the next layer. Figure 8.2 depicts two specific examples of the convolution operations with a layer of size $7 \times 7 \times 1$ and a $3 \times 3 \times 1$ filter in the bottom row. Furthermore, the entire feature map of the next layer is shown on the upper right-hand side of Figure 8.2. Examples of two convolution operations are shown in which the outputs are 16 and 26, respectively. These values are arrived at by using the following multiplication and aggregation operations:

$$5 \times 1 + 8 \times 1 + 1 \times 1 + 1 \times 2 = 16$$
$$4 \times 1 + 4 \times 1 + 4 \times 1 + 7 \times 2 = 26$$

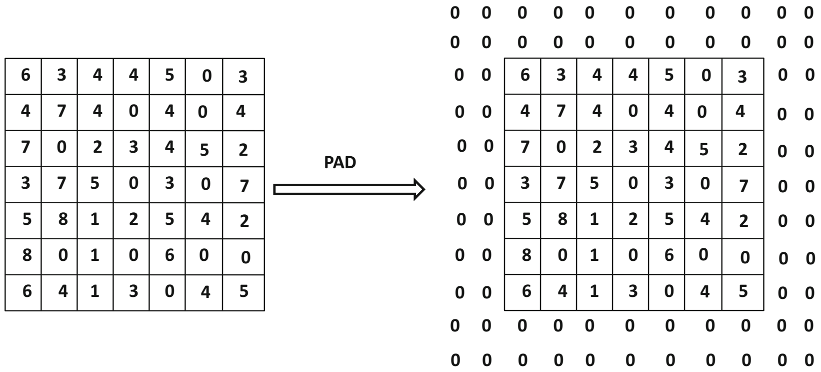
The multiplications with zeros have been omitted in the above aggregation. In the event that the depths of the layer and its corresponding filter are greater than 1, the above operations are performed for each spatial map and then aggregated across the entire depth of the filter.

A convolution in the qth layer increases the *receptive field* of a feature from the qth layer to the $(q+1)$th layer. In other words, each feature in the next layer captures a larger spatial region in the input layer. For example, when using a 3×3 filter convolution successively in three layers, the activations in the first, second, and third hidden layers capture pixel regions of size 3×3, 5×5, and 7×7, respectively, in the *original input image*. As we will see later, other types of operations increase the receptive fields further, as they reduce the size of the spatial footprint of the layers. This is a natural consequence of the fact that features in later layers capture complex characteristics of the image over larger spatial regions, and then combine the simpler features in earlier layers.

When performing the operations from the qth layer to the $(q+1)$th layer, the depth d_{q+1} of the computed layer depends on the *number* of filters in the qth layer, and it is independent of the *depth* of the qth layer or any of its other dimensions. In other words, the depth d_{q+1} in the $(q+1)$th layer is always equal to the number of filters in the qth layer. For example, the depth of the second layer in Figure 8.1(a) is 5, because a total of five filters are used in the first layer for the transformation. However, in order to perform the convolutions in the second layer (to create the third layer), one must now use filters of depth 5 in order to match the new depth of this layer, even though filters of depth 3 were used in the convolutions of the first layer (to create the second layer).

8.2.1 Padding

One observation is that the convolution operation reduces the size of the $(q+1)$th layer in comparison with the size of the qth layer. This type of reduction in size is not desirable in general, because it tends to lose some information along the borders of the image (or of the feature map, in the case of hidden layers). This problem can be resolved by using *padding*. In padding, one adds $(F_q - 1)/2$ "pixels" all around the borders of the feature map in order to maintain the spatial footprint. Note that these pixels are really feature values in the case of padding hidden layers. The value of each of these padded feature values is set



6	3	4	4	5	0	3
4	7	4	0	4	0	4
7	0	2	3	4	5	2
3	7	5	0	3	0	7
5	8	1	2	5	4	2
8	0	1	0	6	0	0
6	4	1	3	0	4	5

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	6	3	4	4	5	0	3	0	0
0	0	4	7	4	0	4	0	4	0	0
0	0	7	0	2	3	4	5	2	0	0
0	0	3	7	5	0	3	0	7	0	0
0	0	5	8	1	2	5	4	2	0	0
0	0	8	0	1	0	6	0	0	0	0
0	0	6	4	1	3	0	4	5	0	0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0

Figure 8.3: An example of padding. Each of the d_q activation maps in the entire depth of the qth layer are padded in this way.

to 0, irrespective of whether the input or the hidden layers are being padded. As a result, the spatial height and width of the input volume will both increase by $(F_q - 1)$, which is exactly what they reduce by (in the output volume) after the convolution is performed. The padded portions do not contribute to the final dot product because their values are set to 0. In a sense, what padding does is to allow the convolution operation with a portion of the filter "sticking out" from the borders of the layer and then performing the dot product only over the portion of the layer where the values are defined. This type of padding is referred to as *half-padding* because (almost) half the filter is sticking out from all sides of the spatial input in the case where the filter is placed in its extreme spatial position along the edges. Half-padding is designed to maintain the spatial footprint exactly.

When padding is not used, the resulting "padding" is also referred to as a *valid padding.* Valid padding generally does not work well from an experimental point of view. Using half-padding ensures that some of the critical information at the borders of the layer is represented in a standalone way. In the case of valid padding, the contributions of the pixels on the borders of the layer will be under-represented compared to the central pixels in the next hidden layer, which is undesirable. Furthermore, this under-representation will be compounded over multiple layers. Therefore, padding is typically performed in all layers, and not just in the first layer where the spatial locations correspond to input values. Consider a situation in which the layer has size $32 \times 32 \times 3$ and the filter is of size $5 \times 5 \times 3$. Therefore, $(5-1)/2 = 2$ zeros are padded on all sides of the image. As a result, the 32×32 spatial footprint first increases to 36×36 because of padding, and then it reduces back to 32×32 after performing the convolution. An example of the padding of a single feature map is shown in Figure 8.3, where two zeros are padded on all sides of the image (or feature map). This is a similar situation as discussed above (in terms of addition of two zeros), except that the spatial dimensions of the image are much smaller than 32×32 in order to enable illustration in a reasonable amount of space.

Another useful form of padding is *full-padding.* In full-padding, we allow (almost) the *full* filter to stick out from various sides of the input. In other words, a portion of the filter of size $F_q - 1$ is allowed to stick out from any side of the input with an overlap of only one spatial feature. For example, the kernel and the input image might overlap at a single pixel at an extreme corner. Therefore, the input is padded with $(F_q - 1)$ zeros on each side. In other words, each spatial dimension of the input increases by $2(F_q - 1)$. Therefore, if the input dimensions in the original image are L_q and B_q, the padded spatial dimensions in the input volume become $L_q + 2(F_q - 1)$ and $B_q + 2(F_q - 1)$. After performing the convolution, the feature-map dimensions in layer $(q+1)$ become $L_q + F_q - 1$ and $B_q + F_q - 1$, respectively. While convolution normally reduces the spatial footprint, full padding *increases* the spatial footprint. Interestingly, full-padding increases each dimension of the spatial footprint by the same value $(F_q - 1)$ that no-padding decreases it. *This relationship is not a coincidence because a "reverse" convolution operation can be implemented by applying another convolution on the fully padded output (of the original convolution) with an appropriately defined kernel of the same size.* This type of "reverse" convolution occurs frequently in the backpropagation and autoencoder algorithms for convolutional neural networks. Fully padded inputs are useful because they increase the spatial footprint, which is required in several types of convolutional autoencoders.

8.2.2 Strides

There are other ways in which convolution can reduce the spatial footprint of the image (or hidden layer). The above approach performs the convolution at every position in the spatial location of the feature map. However, it is not necessary to perform the convolution at every spatial position in the layer. One can reduce the level of granularity of the convolution by using the notion of *strides*. The description above corresponds to the case when a stride of 1 is used. When a stride of S_q is used in the qth layer, the convolution is performed at the locations 1, $S_q + 1$, $2\, S_q + 1$, and so on along both spatial dimensions of the layer. The spatial size of the output on performing this convolution[1] has height of $(L_q - F_q)/S_q + 1$ and a width of $(B_q - F_q)/S_q + 1$. As a result, the use of strides will result in a reduction of each spatial dimension of the layer by a factor of approximately S_q and the area by S_q^2, although the actual factor may vary because of edge effects. It is most common to use a stride of 1, although a stride of 2 is occasionally used as well. It is rare to use strides more than 2 in normal circumstances. Even though a stride of 4 was used in the input layer of the winning architecture [255] of the ILSVRC competition of 2012, the winning entry in the subsequent year reduced the stride to 2 [556] to improve accuracy. Larger strides can be helpful in memory-constrained settings or to reduce overfitting if the spatial resolution is unnecessarily high. Strides have the effect of rapidly increasing the receptive field of each feature in the hidden layer, while reducing the spatial footprint of the entire layer. An increased receptive field is useful in order to capture a complex feature in a larger spatial region of the image. As we will see later, the hierarchical feature engineering process of a convolutional neural network captures more complex shapes in later layers. Historically, the receptive fields have been increased with another operation, known as the *max-pooling* operation. In recent years, larger strides have been used in lieu [184, 466] of max-pooling operations, which will be discussed later.

8.2.3 Typical Settings

It is common to use stride sizes of 1 in most settings. Even when strides are used, small strides of size 2 are used. Furthermore, it is common to have $L_q = B_q$. In other words, it is desirable to work with square images. In cases where the input images are not square, preprocessing is used to enforce this property. For example, one can extract square patches

[1] Here, it is assumed that $(L_q - F_q)$ is exactly divisible by S_q in order to obtain a clean fit of the convolution filter with the original image. Otherwise, some ad hoc modifications are needed to handle edge effects. In general, this is not a desirable solution.

of the image to create the training data. The number of filters in each layer is often a power of 2, because this often results in more efficient processing. Such an approach also leads to hidden layer depths that are powers of 2. Typical values of the spatial extent of the filter size (denoted by F_q) are 3 or 5. In general, small filter sizes often provide the best results, although some practical challenges exist in using filter sizes that are too small. Small filter sizes typically lead to deeper networks (for the same parameter footprint) and therefore tend to be more powerful. In fact, one of the top entries in an ILSVRC contest, referred to as *VGG* [454], was the first to experiment with a spatial filter dimension of only $F_q = 3$ for all layers, and the approach was found to work very well in comparison with larger filter sizes.

Use of Bias

As in all neural networks, it is also possible to add biases to the forward operations. Each unique filter in a layer is associated with its own bias. Therefore, the pth filter in the qth layer has bias $b^{(p,q)}$. When any convolution is performed with the pth filter in the qth layer, the value of $b^{(p,q)}$ is added to the dot product. The use of the bias simply increases the number of parameters in each filter by 1, and therefore it is not a significant overhead. Like all other parameters, the bias is learned during backpropagation. One can treat the bias as a weight of a connection whose input is always set to +1. This special input is used in all convolutions, irrespective of the spatial location of the convolution. Therefore, one can assume that a special pixel appears in the input whose value is always set to 1. Therefore, the number of input features in the qth layer is $1 + L_q \times B_q \times d_q$. This is a standard feature-engineering trick that is used for handling bias in all forms of machine learning.

8.2.4 The ReLU Layer

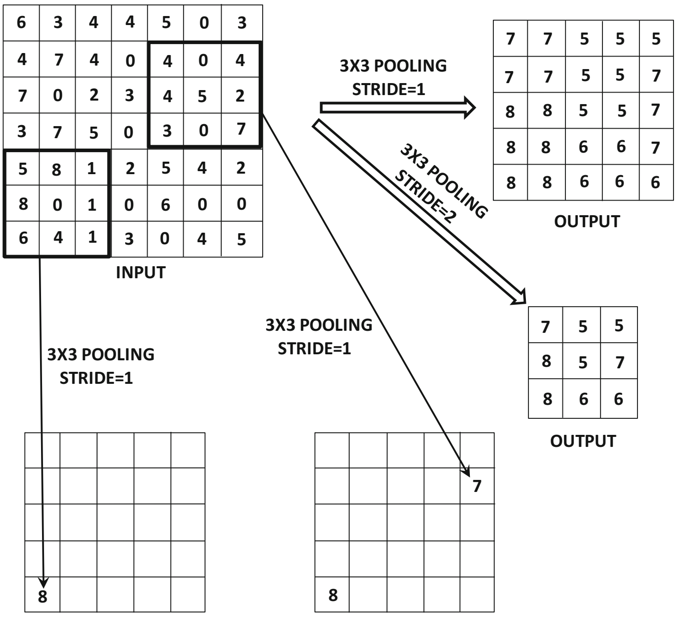
The convolution operation is interleaved with the pooling and ReLU operations. The ReLU activation is not very different from how it is applied in a traditional neural network. For each of the $L_q \times B_q \times d_q$ values in a layer, the ReLU activation function is applied to it to create $L_q \times B_q \times d_q$ thresholded values. These values are then passed on to the next layer. Therefore, applying the ReLU does not change the dimensions of a layer because it is a simple one-to-one mapping of activation values. In traditional neural networks, the activation function is combined with a linear transformation with a matrix of weights to create the next layer of activations. Similarly, a ReLU typically follows a convolution operation (which is the rough equivalent of the linear transformation in traditional neural networks), and the ReLU layer is often not explicitly shown in pictorial illustrations of the convolution neural network architectures.

It is noteworthy that the use of the ReLU activation function is a recent evolution in neural network design. In the earlier years, saturating activation functions like sigmoid and tanh were used. However, it was shown in [255] that the use of the ReLU has tremendous advantages over these activation functions both in terms of speed and accuracy. Increased speed is also connected to accuracy because it allows one to use deeper models and train them for a longer time. In recent years, the use of the ReLU activation function has replaced the other activation functions in convolutional neural network design to an extent that this chapter will simply use the ReLU as the default activation function (unless otherwise mentioned).

8.2.5 Pooling

The pooling operation is, however, quite different. The pooling operation works on small grid regions of size $P_q \times P_q$ in each layer, and produces another layer *with the same depth* (unlike filters). For each square region of size $P_q \times P_q$ in each of the d_q activation maps, the *maximum* of these values is returned. This approach is referred to as *max-pooling.* If a stride of 1 is used, then this will produce a new layer of size $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$. However, it is more common to use a stride $S_q > 1$ in pooling. In such cases, the length of the new layer will be $(L_q - P_q)/S_q + 1$ and the breadth will be $(B_q - P_q)/S_q + 1$. Therefore, pooling drastically reduces the spatial dimensions of each activation map.

Unlike with convolution operations, pooling is done at the level of *each* activation map. Whereas a convolution operation simultaneously uses all d_q feature maps in combination with a filter to produce a single feature value, pooling independently operates on each feature map to produce another feature map. Therefore, the operation of pooling does not change the number of feature maps. In other words, the depth of the layer created using pooling is the same as that of the layer on which the pooling operation was performed. Examples of pooling with strides of 1 and 2 are shown in Figure 8.4. Here, we use pooling over 3×3 regions. The typical size P_q of the region over which one performs pooling is 2×2. At a stride of 2, there would be no overlap among the different regions being pooled, and it is quite common to use this type of setting. However, it has sometimes been suggested that it is desirable to have at least some overlap among the spatial units at which the pooling is performed, because it makes the approach less likely to overfit.



INPUT

6	3	4	4	5	0	3
4	7	4	0	4	0	4
7	0	2	3	4	5	2
3	7	5	0	3	0	7
5	8	1	2	5	4	2
8	0	1	0	6	0	0
6	4	1	3	0	4	5

OUTPUT

7	7	5	5	5
7	7	5	5	7
8	8	5	5	7
8	8	6	6	7
8	8	6	6	6

OUTPUT

7	5	5
8	5	7
8	6	6

8				

				7
8				

Figure 8.4: An example of a max-pooling of one activation map of size 7×7 with strides of 1 and 2. A stride of 1 creates a 5×5 activation map with heavily repeating elements because of maximization in overlapping regions. A stride of 2 creates a 3×3 activation map with less overlap. Unlike convolution, each activation map is independently processed and therefore the number of output activation maps is exactly equal to the number of input activation maps.

Other types of pooling (like average-pooling) are possible but rarely used. In the earliest convolutional network, referred to as *LeNet-5*, a variant of average pooling was used and was referred[2] to as *subsampling*. In general, max-pooling remains more popular than average pooling. The max-pooling layers are interleaved with the convolutional/ReLU layers, although the former typically occurs much less frequently in deep architectures. This is because pooling drastically reduces the spatial size of the feature map, and only a few pooling operations are required to reduce the spatial map to a small constant size.

It is common to use pooling with 2×2 filters and a stride of 2, when it is desired to reduce the spatial footprint of the activation maps. Pooling results in (some) invariance to translation because shifting the image slightly does not change the activation map significantly. This property is referred to as *translation invariance*. The idea is that similar images often have very different relative locations of the distinctive shapes within them, and translation invariance helps in being able to classify such images in a similar way. For example, one should be able to classify a bird as a bird, irrespective of where it occurs in the image.

Another important purpose of pooling is that it increases the size of the receptive field while reducing the spatial footprint of the layer because of the use of strides larger than 1. Increased sizes of receptive fields are needed to be able to capture larger regions of the image within a complex feature in later layers. Most of the rapid reductions in spatial footprints of the layers (and corresponding increases in receptive fields of the features) are caused by the pooling operations. Convolutions increase the receptive field only gently unless the stride is larger than 1. In recent years, it has been suggested that pooling is not always necessary. One can design a network with only convolutional and ReLU operations, and obtain the expansion of the receptive field by using larger strides within the convolutional operations [184, 466]. Therefore, there is an emerging trend in recent years to get rid of the max-pooling layers altogether. However, this trend has not been fully established and validated, as of the writing of this book. There seem to be at least some arguments in favor of max-pooling. Max-pooling introduces nonlinearity and a greater amount of translation invariance, as compared to strided convolutions. Although nonlinearity can be achieved with the ReLU activation function, the key point is that the effects of max-pooling cannot be exactly replicated by strided convolutions either. At the very least, the two operations are not fully interchangeable.

8.2.6 Fully Connected Layers

Each feature in the final spatial layer is connected to each hidden state in the first fully connected layer. This layer functions in exactly the same way as a traditional feed-forward network. In most cases, one might use more than one fully connected layer to increase the power of the computations towards the end. The connections among these layers are exactly structured like a traditional feed-forward network. Since the fully connected layers are densely connected, the vast majority of parameters lie in the fully connected layers. For example, if each of two fully connected layers has 4096 hidden units, then the connections between them have more than 16 million weights. Similarly, the connections from the last spatial layer to the first fully connected layer will have a large number of parameters. Even though the convolutional layers have a larger number of *activations* (and a larger memory footprint), the fully connected layers often have a larger number of *connections* (and parameter footprint). The reason that activations contribute to the memory footprint

[2] In recent years, subsampling also refers to other operations that reduce the spatial footprint. Therefore, there is some difference between the classical usage of this term and modern usage.

more significantly is that the number of activations are multiplied by mini-batch size while tracking variables in the forward and backward passes of backpropagation. These trade-offs are useful to keep in mind while choosing neural-network design based on specific types of resource constraints (e.g., data versus memory availability). It is noteworthy that the nature of the fully-connected layer can be sensitive to the application at hand. For example, the nature of the fully-connected layer for a classification application would be somewhat different from the case of a segmentation application. The aforementioned discussion is for the most common use-case of a classification application.

The output layer of a convolutional neural network is designed in an application-specific way. In the following, we will consider the representative application of classification. In such a case, the output layer is fully connected to every neuron in the penultimate layer, and has a weight associated with it. One might use the logistic, softmax, or linear activation depending on the nature of the application (e.g., classification or regression).

One alternative to using fully connected layers is to use average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final spatial layer will be exactly equal to the number of filters. In this scenario, if the final activation maps are of size $7 \times 7 \times 256$, then 256 features will be created. Each feature will be the result of aggregating 49 values. This type of approach greatly reduces the parameter footprint of the fully connected layers, and it has some advantages in terms of generalizability. This approach was used in *GoogLeNet* [485]. In some applications like image segmentation, each pixel is associated with a class label, and one does not use fully connected layers. Fully convolutional networks with 1×1 convolutions are used in order to create an output spatial map.

8.2.7 The Interleaving Between Layers

The convolution, pooling, and ReLU layers are typically interleaved in a neural network in order to increase the expressive power of the network. The ReLU layers often follow the convolutional layers, just as a nonlinear activation function typically follows the linear dot product in traditional neural networks. Therefore, the convolutional and ReLU layers are typically stuck together one after the other. Some pictorial illustrations of neural architectures like *AlexNet* [255] do not explicitly show the ReLU layers because they are assumed to be always stuck to the end of the linear convolutional layers. After two or three sets of convolutional-ReLU combinations, one might have a max-pooling layer. Examples of this basic pattern are as follows:

$$\text{CRCRP}$$
$$\text{CRCRCRP}$$

Here, the convolutional layer is denoted by C, the ReLU layer is denoted by R, and the max-pooling layer is denoted by P. This entire pattern (including the max-pooling layer) might be repeated a few times in order to create a deep neural network. For example, if the first pattern above is repeated three times and followed by a fully connected layer (denoted by F), then we have the following neural network:

$$\text{CRCRPCRCRPCRCRPF}$$

The description above is not complete because one needs to specify the number/size/padding of filters/pooling layers. The pooling layer is the key step that tends to reduce the spatial footprint of the activation maps because it uses strides that are larger than 1. It is also possible to reduce the spatial footprints with strided convolutions instead of max-pooling. These networks are often quite deep, and it is not uncommon to have convolutional networks with more than 15 layers. Recent architectures also use *skip connections* between layers, which become increasingly important as the depth of the network increases (cf. Section 8.4.5).

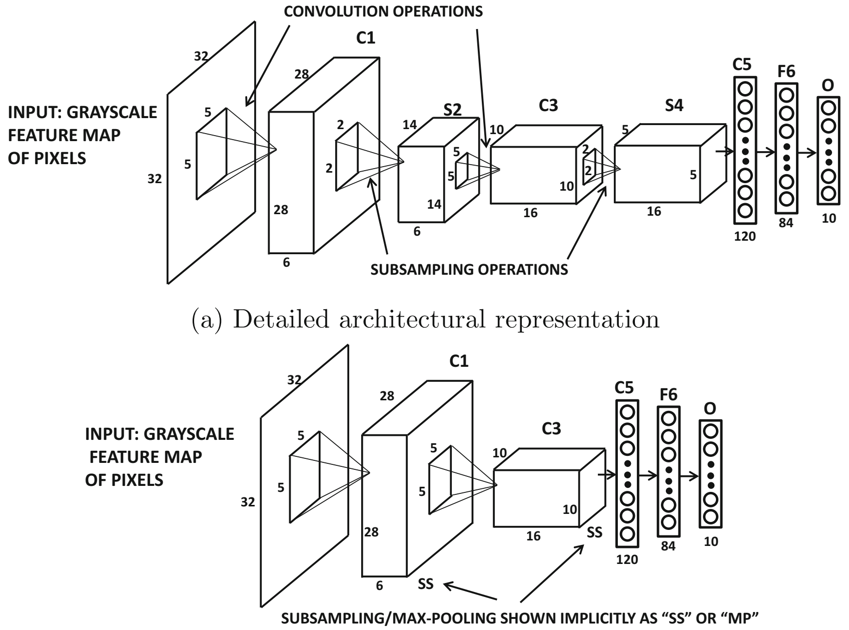
LeNet-5

Early networks were quite shallow. An example of one of the earliest neural networks is *LeNet-5* [279]. The input data is in grayscale, and there is only one color channel. The input is assumed to be the ASCII representation of a character. For the purpose of discussion, we will assume that there are ten types of characters (and therefore 10 outputs), although the approach can be used for any number of classes.

The network contained two convolution layers, two pooling layers, and three fully connected layers at the end. However, later layers contain multiple feature maps because of the use of multiple filters in each layer. The architecture of this network is shown in Figure 8.5. The first fully connected layer was also referred to as a convolution layer (labeled as $C5$) in the original work because the ability existed to generalize it to spatial features for larger input maps. However, the specific implementation of *LeNet-5* really used $C5$ as a fully connected layer, because the filter spatial size was the same as the input spatial size. This is why we are counting $C5$ as a fully connected layer in this exposition. It is noteworthy that two versions of *LeNet-5* are shown in Figure 8.5(a) and (b). The upper diagram of Figure 8.5(a) explicitly shows the subsampling layers, which is how the architecture was presented in the original work. However, deeper architectural diagrams like *AlexNet* [255] often do not show the subsampling or max-pooling layers explicitly in order to accommodate the large number of layers. Such a concise architecture for *LeNet-5* is illustrated in Figure 8.5(b). The activation function layers are also not explicitly shown in either figure. In the original work in *LeNet-5*, the sigmoid activation function occurs immediately after the subsampling operations, although this ordering is relatively unusual in recent architectures. In most modern architectures, subsampling is replaced by max-pooling, and the max-pooling layers occur less frequently than the convolution layers. Furthermore, the activations are typically performed immediately after each convolution (rather than after each max-pooling).

The number of layers in the architecture is often counted in terms of the number of layers with weighted spatial filters and the number of fully connected layers. In other words, subsampling/max-pooling and activation function layers are often not counted separately. The subsampling in *LeNet-5* used 2 × 2 spatial regions with stride 2. Furthermore, unlike max-pooling, the values were averaged, scaled with a trainable weight and then a bias was added. In modern architectures, the linear scaling and bias addition operations have been dispensed with. The concise architectural representation of Figure 8.5(b) is sometimes confusing to beginners because it is missing details such as the size of the max-pooling/subsampling filters. In fact, there is no unique way of representing these architectural details, and many variations are used by different authors. This chapter will show several such examples in the case studies.

This network is extremely shallow by modern standards; yet the basic principles have not changed since then. The main difference is that the ReLU activation had not appeared at that point, and sigmoid activation was often used in the earlier architectures. Furthermore, the use of average pooling is extremely uncommon today compared to max-pooling. Recent years have seen a move away from both max-pooling and subsampling, with strided convolutions as the preferred choice. *LeNet-5* also used ten radial basis function (RBF) units in the final layer (cf. Chapter 5), in which the prototype of each unit was compared to its input vector and the squared Euclidean distance between them was output. This is the same as using the negative log-likelihood of the Gaussian distribution represented by that RBF unit. The parameter vectors of the RBF units were chosen by hand, and correspond to a stylized 7×12 bitmap image of the corresponding character class, which were flattened into a $7 \times 12 = 84$-dimensional representation. Note that the size of the penultimate layer is exactly 84 in order to enable the computation of the Euclidean distance between the vector corresponding to that layer and the parameter vector of the RBF unit. The ten outputs in the final layer provide the scores of the classes, and the smallest score among the ten units provides the prediction. This type of use of RBF units is now anachronistic in modern convolutional network design, and one generally tends to work with softmax units with log-likelihood loss on multinomial label outputs. *LeNet-5* was used extensively for character recognition, and was used by many banks to read checks.



(a) Detailed architectural representation

(b) Concise architectural representation

Figure 8.5: LeNet-5: One of the earliest convolutional neural networks.

8.2.8 Local Response Normalization

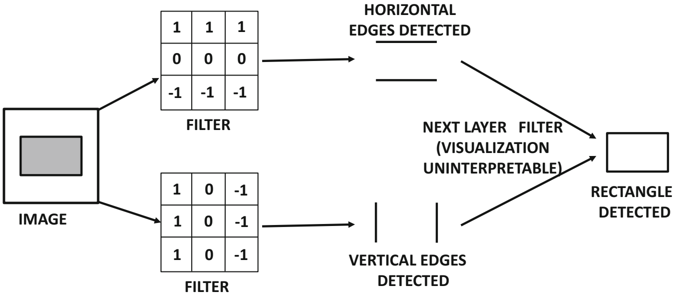
A trick that is introduced in [255] is that of *local response normalization*, which is always used immediately after the ReLU layer. The use of this trick aids generalization. The basic idea of this normalization approach is inspired from biological principles, and it is intended to create competition among different filters. First, we describe the normalization formula using *all* filters, and then we describe how it is actually computed using only a subset of filters. Consider a situation in which a layer contains N filters, and the activation values of these N filters at a particular spatial position (x, y) are given by $a_1 \dots a_N$. Then, each a_i is converted into a normalized value b_i using the following formula:

$$b_i = \frac{a_i}{(k + \alpha \sum_j a_i^2)^\beta} \tag{8.1}$$

The values of the underlying parameters used in [255] are $k = 2$, $\alpha = 10^{-4}$, and $\beta = 0.75$. However, in practice, one does not normalize over all N filters. Rather the filters are ordered arbitrarily up front to define "adjacency" among filters. Then, the normalization is performed over each set of n "adjacent" filters for some parameter n. The value of n used in [255] is 5. Therefore, we have the following formula:

$$b_i = \frac{a_i}{(k + \alpha \sum_{j=i-\lfloor n/2 \rfloor}^{i+\lfloor n/2 \rfloor} a_i^2)^\beta} \tag{8.2}$$

In the above formula, any value of $i - n/2$ that is less than 0 is set to 0, and any value of $i + n/2$ that is greater than N is set to N. The use of this type of normalization is no obsolete, and its discussion has been included here for historical reasons.


HORIZONTAL EDGES DETECTED
FILTER
NEXT LAYER FILTER (VISUALIZATION UNINTERPRETABLE)
IMAGE
RECTANGLE DETECTED
FILTER
VERTICAL EDGES DETECTED

1	1	1
0	0	0
-1	-1	-1

1	0	-1
1	0	-1
1	0	-1

Figure 8.6: Filters detect edges and combine them to create rectangle.

8.2.9 Hierarchical Feature Engineering

It is instructive to examine the activations of the filters created by real-world images in different layers. In Section 8.5, we will discuss a concrete way in which the features extracted in various layers can be visualized. For now, we provide a subjective interpretation. The activations of the filters in the early layers are low-level features like edges, whereas those in later layers put together these low-level features. For example, a mid-level feature might put together edges to create a hexagon, whereas a higher-level feature might put together the mid-level hexagons to create a honeycomb. It is fairly easy to see why a low-level filter might detect edges. Consider a situation in which the color of the image changes along an edge. As a result, the difference between neighboring pixel values will be non-zero only across the edge. This can be achieved by choosing the appropriate weights in the corresponding low-level filter. Note that the filter to detect a horizontal edge will not be the same as that to detect a vertical edge. This brings us back to Hubel and Weisel's experiments in which different neurons in the cat's visual cortex were activated by different edges. Examples of filters detecting horizontal and vertical edges are illustrated in Figure 8.6. The next layer filter works on the hidden features and therefore it is harder to interpret. Nevertheless, the next layer filter is able to detect a rectangle by combining the horizontal and vertical edges.

In a later section, we will show visualizations of how smaller portions of real-world image activate different hidden features, much like the biological model of Hubel and Wiesel in which different shapes seem to activate different neurons. Therefore, the power of convolutional neural networks rests in the ability to put together these primitive shapes into more complex shapes layer by layer. Note that it is impossible for the first convolution layer to learn any feature that is larger than $F_1 \times F_1$ pixels, where the value of F_1 is typically a small number like 3 or 5. However, the next convolution layer will be able to put together many of these patches together to create a feature from an area of the image that is larger. The primitive features learned in earlier layers are put together in a semantically coherent way to learn increasingly complex and interpretable visual features. The choice of learned features is affected by how backpropagation adapts the features to the needs of the loss function at hand. For example, if an application is training to classify images as cars, the approach might learn to put together arcs to create a circle, and then it might put together circles with other shapes to create a car wheel. All this is enabled by the hierarchical features of a deep network.

Recent *ImageNet* competitions have demonstrated that much of the power in image recognition lies in increased depth of the network. Not having enough layers effectively prevents the network from learning the hierarchical regularities in the image that are combined to create its semantically relevant components. Another important observation is that the nature of the features learned will be sensitive to the specific data set at hand. For example, the features learned to recognize trucks will be different from those learned to recognize carrots. However, some data sets (like *ImageNet*) are diverse enough that the features learned by training on these data sets have general-purpose significance across many applications.

8.3 Training a Convolutional Network

The process of training a convolutional neural network uses the backpropagation algorithm. There are primarily three types of layers, corresponding to the convolution, ReLU, and max-pooling layers. We will separately describe the backpropagation algorithm through each of these layers. The ReLU is relatively straightforward to backpropagate through because it is no different than a traditional neural network. For max-pooling with no overlap between pools, one only needs to identify which unit is the maximum value in a pool (with ties broken arbitrarily or divided proportionally). The partial derivative of the loss with respect to the pooled state flows back to the unit with maximum value. All entries other than the maximum entry in the grid will be assigned a value of 0. Note that the backpropagation through a maximization operation is also described in Table 3.1 of Chapter 3. For cases in which the pools are overlapping, let $P_1 \ldots P_r$ be the pools in which the unit h is involved, with corresponding activations $h_1 \ldots h_r$ in the next layer. If h is the maximum value in pool P_i (and therefore $h_i = h$), then the gradient of the loss with respect to h_i flows back to h (with ties broken arbitrarily or divided proportionally). The contributions of the different overlapping pools (from $h_1 \ldots h_r$ in the next layer) are added in order to compute the gradient with respect to the unit h. Therefore, the backpropagation through the maximization and the ReLU operations are not very different from those in traditional neural networks.

8.3.1 Backpropagating Through Convolutions

The backpropagation through convolutions is also not very different from the backpropagation with linear transformations (i.e., matrix multiplications) in a feed-forward network. This point of view will become particularly clear when we present convolutions as a form of matrix multiplication. Just as backpropagation in feed-forward networks from layer $(i + 1)$ to layer i is achieved by multiplying the error derivatives with respect to layer $(i + 1)$ with the transpose of the forward propagation matrix between layers i and $(i + 1)$ (cf. Table 3.1 of Chapter 3), backpropagation in convolutional networks can also be seen as a form of transposed convolution.

First, we describe a simple element-wise approach to backpropagation. Assume that the loss gradients of the cells in layer $(i + 1)$ have already been computed. The loss derivative with respect to a cell in layer $(i + 1)$ is defined as the partial derivative of the loss function with respect to the hidden variable in that cell. Convolutions multiply the activations in layer i with filter elements to create elements in the next layer. Therefore, a cell in layer $(i + 1)$ receives aggregated contributions from a 3-dimensional volume of elements in the previous layer of filter size $F_i \times F_i \times d_i$. At the same time, a cell c in layer i contributes to multiple elements (denoted by set S_c) in layer $(i + 1)$, although the number of elements to which it contributes depends on the depth of the next layer and the stride. Identifying this "forward set" is the key to the backpropagation. A key point is that the cell c contributes to each element in S_c in an additive way after multiplying the activation of cell c with a filter element. Therefore, backpropagation simply needs to multiply the loss derivative of each element in S_c with respect to the corresponding filter element and aggregate in the backwards direction at c. For any particular cell c in layer i, the following pseudocode can be used to backpropagate the existing derivatives in layer-$(i+1)$ to cell c in layer-i:

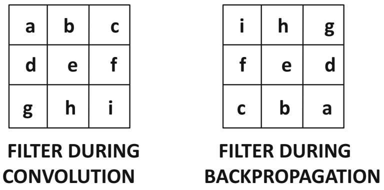
```
Identify all cells $S_c$ in layer $(i + 1)$ to which cell $c$ in layer $i$ contributes;
For each cell $r \in S_c$, let $\delta_r$ be its (already backpropagated) loss-derivative with respect to cell $r$;
For each cell $r \in S_c$, let $w_r$ be weight of filter element used for contributing from cell $c$ to $r$;
$\delta_c = \sum_{r \in S_c} \delta_r \cdot w_r$;
```

After the loss gradients have been computed, the values are multiplied with those of the hidden units of the $(i - 1)$th layer to obtain the gradients with respect to the weights between the $(i - 1)$th and ith layer. In other words, the hidden value at one end point of a weight is multiplied with the loss gradient at the other end in order to obtain the partial derivative with respect to the weight. However, this computation assumes that all weights are distinct, whereas the weights in the filter are shared across the entire spatial extent of the layer. Therefore, one has to be careful to account for shared weights, and sum up the partial derivatives of all copies of a shared weight. In other words, we first pretend that the filter used in each position is distinct in order to compute the partial derivative with respect to each copy of the shared weight, and then add up the partial derivatives of the loss with respect to all copies of a particular weight.

Note that the approach above uses simple linear accumulation of gradients like traditional backpropagation. However, one has to be slightly careful in terms of keeping track of the cells that influence other cells in the next layer. One can implement backpropagation with the help of tensor multiplication operations, which can further be simplified into simple matrix multiplications of derived matrices from these tensors. This point of view will be discussed in the next two sections because it provides many insights on how many aspects of feedforward networks can be generalized to convolutional neural networks.

8.3.2 Backpropagation as Convolution with Inverted/Transposed Filter

In conventional neural networks, a backpropagation operation is performed by multiplying a vector of gradients at layer $(q+1)$ with the transposed weight matrix between the layers q and $(q+1)$ in order to obtain the vector of gradients at layer q (cf. Table 3.1). In convolution neural networks, the backpropagated derivatives are also associated with spatial positions in the layers. Is there an analogous convolution we can apply to the spatial footprint of backpropagated derivatives in a layer to obtain those of the previous layer? It turns out that this is indeed possible.



a	b	c
d	e	f
g	h	i

FILTER DURING CONVOLUTION

i	h	g
f	e	d
c	b	a

FILTER DURING BACKPROPAGATION

Figure 8.7: The inverse of a kernel for backpropagation

Let us consider the case in which the activations in layer q are convolved with a filter to create those in layer $(q+1)$. For simplicity, consider the case in which depth d_q of the input layer and the depth d_{q+1} of the output layer are both 1; furthermore, we use convolutions with stride 1. In such a case, the convolution filter is inverted both horizontally and vertically for backpropagation. An example of such an inverted filter is illustrated in Figure 8.7. The intuitive reason for this inversion is that the *filter* is "moved around" the spatial area of the input volume to perform the dot product, whereas the backpropagated derivatives are with respect to the *input volume*, whose relative movement with respect to the filter is the opposite of the filter movement during convolutions. Note that the entry in the extreme upper-left of the convolution filter might not even contribute to the extreme upper-left entry in the output volume (because of padding), but it will almost always contribute to the extreme lower-right entry of the output volume. This is consistent with the inversion of the filter. The backpropagated derivative set of the $(q+1)$th layer is convolved with this inverted filter to obtain the backpropagated derivative set of the qth layer. How are the paddings of the forward convolution and backward convolution related? For a stride of 1, the sum of the paddings during forward propagation and backward propagation is $F_q - 1$, where F_q is the side length of the filter for qth layer.

Now consider the case in which the depths d_q and d_{q+1} are no longer 1, but are arbitrary values. In this case, an additional tensor transposition needs to occur. The weight of the (i, j, k)th position of the pth filter in the qth layer is $\mathcal{W} = [w_{ijk}^{(p,q)}]$. Note that i and j refer to spatial positions, whereas k refers to the depth-centric position of the weight. In such a case, let the 5-dimensional tensor corresponding to the backpropagation filters from layer $q+1$ to layer q be denoted by $\mathcal{U} = [u_{ijk}^{(p,q+1)}]$. Then, the entries of this tensor are as follows:

$$u_{rsp}^{(k,q+1)} = w_{ijk}^{(p,q)} \tag{8.3}$$

Here, we have $r = F_q - i + 1$ and $s = F_q - j + 1$. Note that the index p of the filter identifier and depth k within a filter have been interchanged between $\mathcal{W}$ and $\mathcal{U}$ in Equation 8.3. This is a tensor-centric transposition.

In order to understand the transposition above, consider a situation in which we use 20 filters on the 3-channel RGB volume in order to create an output volume of depth 20. While backpropagating, we will need to take a *gradient volume* of depth 20 and transform to a *gradient volume* of depth 3. Therefore, we need to create 3 filters for backpropagation, each of which is for the red, green, and blue colors. We pull out the 20 spatial slices from the 20 filters that are applied to the red color, invert them using the approach of Figure 8.7, and then create a single 20-depth filter for backpropagating gradients with respect to the red slice. Similar approaches are used for the green and blue slices. The transposition and inversion in Equation 8.3 correspond to these operations.

8.3.3 Convolution/Backpropagation as Matrix Multiplications

It is helpful to view convolution as a matrix multiplication because it helps us define various related notions such as *transposed convolution*, *deconvolution*, and *fractional convolution*. These concepts are helpful not just in understanding backpropagation, but also in developing the machinery necessary for convolutional autoencoders. In traditional feed-forward networks, matrices that are used to transform hidden states in the forward phase are transposed in the backwards phase (cf. Table 3.1) in order to backpropagate partial derivatives across layers. Similarly, the matrices used in encoders are often transposed in the decoders when working with autoencoders in traditional settings. Although the spatial structure of the convolutional neural network does mask the nature of the underlying matrix multiplication, one can "flatten" this spatial structure to perform the multiplication and reshape back to a spatial structure using the known spatial positions of the elements of the flattened matrix. This somewhat indirect approach is helpful in understanding the fact that the convolution operation is similar to the matrix multiplication in feed-forward networks at a very fundamental level. Furthermore, real-world implementations of convolution are often accomplished with matrix multiplication.

For simplicity, let us first consider the case in which the qth layer and the corresponding filter used for convolution both have unit depth. Furthermore, assume that we are using a stride of 1 with zero padding. Therefore, the input dimensions are $L_q \times B_q \times 1$, and the output dimensions are $(L_q - F_q + 1) \times (B_q - F_q + 1) \times 1$. In the common setting in which the spatial dimensions are square (i.e., $L_q = B_q$), one can assume that the spatial dimensions of the input $A_I = L_q \times L_q$ and the spatial dimensions of the output are $A_O = (L_q - F_q + 1) \times (L_q - F_q + 1)$. Here, A_I and A_O are the spatial areas of the input and output matrices, respectively. The input can be representing by flatting the area A_I into an A_I-dimensional column vector in which the rows of the spatial area are concatenated from top to bottom. This vector is denoted by $\overline{f}$. An example of a case in which we use a 2×2 filter on a 3×3 input is shown in Figure 8.8. Therefore, the output is of size 2×2, and we have $A_I = 3 \times 3 = 9$, and $A_O = 2 \times 2 = 4$. The 9-dimensional column vector for the 3×3 input is shown in Figure 8.8. A sparse matrix C is defined in lieu of the filter, which is the key in representing the convolution as a matrix multiplication. A matrix of size $A_O \times A_I$ is defined in which each row corresponds to the convolution at one of the A_O convolution locations. These rows are associated with the spatial location of the top-left corner of the convolution region in the input matrix from which they were derived. The value of each entry in the row corresponds to one of the A_I positions in the input matrix, but this value is 0, if that input position is not involved in the convolution for that row. Otherwise, the value is set to the corresponding value of the filter, which is used for multiplication. The ordering of the entries in a row is based on the same spatially sensitive ordering of the input matrix locations as was used to flatten the input matrix into an A_I-dimensional vector. Since the filter size is usually much smaller than the input size, most of the entries in the matrix C are 0s, and each entry of the filter occurs in every row of C. Therefore, every entry in the filter is repeated A_O times in C, because it is used for A_O multiplications.

An example of a 4×9 matrix C is shown in Figure 8.8. Subsequent multiplication of C with $\overline{f}$ yields an A_O-dimensional vector. The corresponding 4-dimensional vector is shown in Figure 8.8. Since each of the A_O rows of C is associated with a spatial location, these locations are inherited by $C\overline{f}$. These spatial locations are used to reshape $C\overline{f}$ to a spatial matrix. The reshaping of the 4-dimensional vector to a 2×2 matrix is also shown in Figure 8.8.

This particular exposition uses the simplified case with a depth of 1. In the event that the depth is larger than 1, the same approach is applied for each 2-dimensional slice, and the results are added. In other words, we aggregate $\sum_p C_p \overline{f}_p$ over the various slice indices p and then the results are re-shaped into a 2-dimensional matrix. This approach amounts to a *tensor* multiplication, which is a straightforward generalization of a matrix multiplication. The tensor multiplication approach is how convolution is actually implemented in practice. In general, one will have multiple filters, which correspond to multiple output maps. In such a case, the kth filter will be converted into the sparsified matrix $C_{p,k}$, and the kth feature map of the output volume will be $\sum_p C_{p,k} \overline{f}_p$.

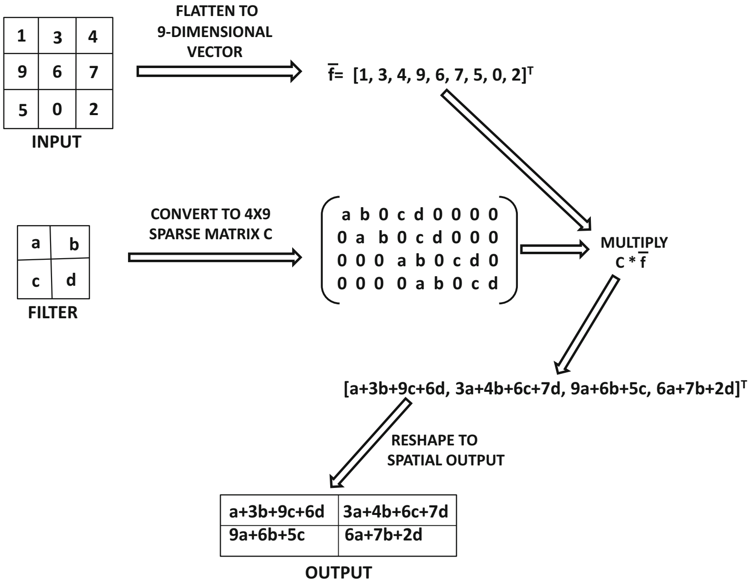

1 3 4
9 6 7
5 0 2
INPUT
FLATTEN TO 9-DIMENSIONAL VECTOR
f̄= [1, 3, 4, 9, 6, 7, 5, 0, 2]ᵀ
a b
c d
FILTER
CONVERT TO 4X9 SPARSE MATRIX C
a b 0 c d 0 0 0 0
0 a b 0 c d 0 0 0
0 0 0 a b 0 c d 0
0 0 0 0 a b 0 c d
MULTIPLY C * f̄
[a+3b+9c+6d, 3a+4b+6c+7d, 9a+6b+5c, 6a+7b+2d]ᵀ
RESHAPE TO SPATIAL OUTPUT
a+3b+9c+6d 3a+4b+6c+7d
9a+6b+5c 6a+7b+2d
OUTPUT

Figure 8.8: Convolution as matrix multiplication

The matrix-centric approach is very useful for performing backpropagation because one can also propagate gradients backwards by using the same approach in the backwards direction, except that the *transposed* matrix C^T is used for multiplication with the flattened vector version of a 2-dimensional slice of the output gradient. Note that the flattening of a gradient with respect to a spatial map can be done in a similar way as the flattened vector $\overline{f}$ is created in the forward phase. Consider the simple case in which both the input and output volumes have a depth of 1. If $\overline{g}$ is the flattened vector gradient of the loss with respect to the output spatial map, then the flattened gradient with respect to the input spatial map is obtained as $C^T\overline{g}$. This approach is consistent with the approach used in feed-forward networks, in which the transpose of the forward matrix is used in backpropagation. The above result is for the simple case when both input and output volumes have depth of 1. What happens in the general case? When the depth of the output volume is $d > 1$, the gradients with respect to the output maps are denoted by $\overline{g}_1 \ldots \overline{g}_d$. The corresponding gradient with respect to the features in the pth spatial slice of the input volume is given by $\sum_{k=1}^{d} C_{p,k}^T \overline{g}_k$. Here, the matrix $C_{p,k}$ is obtained by converting the pth spatial slice of the kth filter into the sparsified matrix as discussed above. This approach is a consequence of Equation 8.3. This type of transposed convolution is also useful for the deconvolution operation in convolution autoencoders, which will be discussed later in this chapter (cf. Section 8.5).

8.3.4 Data Augmentation

A common trick to reduce overfitting in convolutional neural networks is the idea of *data augmentation*. In data augmentation, new training examples are generated by using transformations on the original examples. This idea was briefly discussed in Chapter 4, although it works better in some domains than others. Image processing is one domain to which data augmentation is very well suited. This is because many transformations such as translation, rotation, patch extraction, and reflection do not fundamentally change the properties of the object in an image. However, they do increase the generalization power of the data set when trained with the augmented data set. For example, if a data set is trained with mirror images and reflected versions of all the bananas in it, then the model is able to better recognize bananas in different orientations.

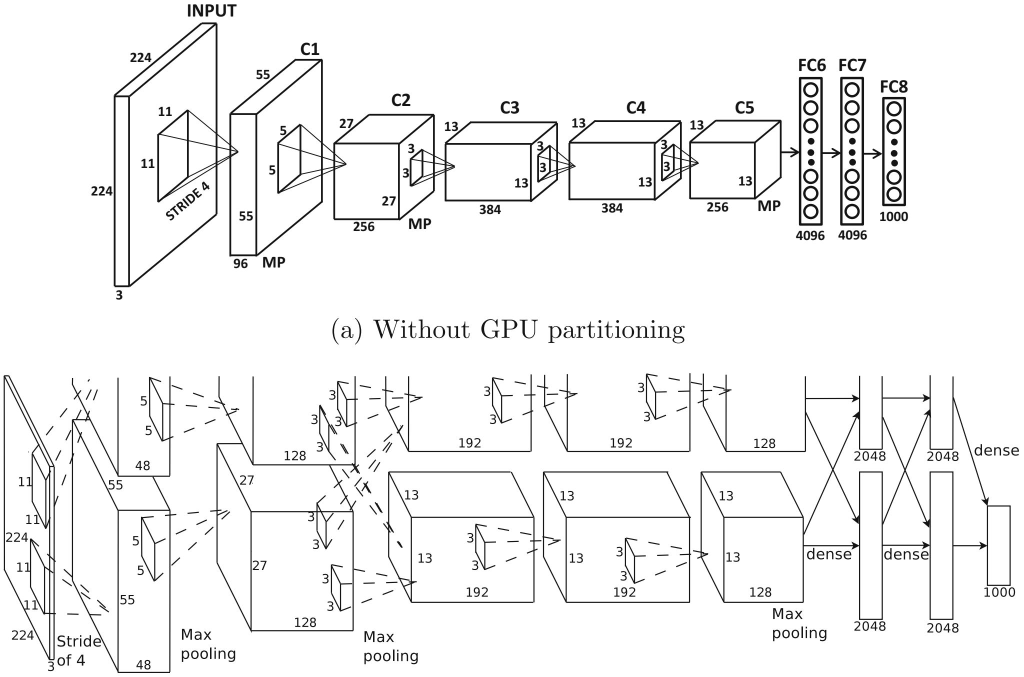
Many of these forms of data augmentation require very little computation, and therefore the augmented images do not need to be explicitly generated up front. Rather, they can be created at training time, when an image is being processed. For example, while processing an image of a banana, it can be reflected into a modified banana at training time. Similarly, the same banana might be represented in somewhat different color intensities in different images, and therefore it might be helpful to create representations of the same image in different color intensities. In many cases, creating the training data set using image patches can be helpful. An important neural network that rekindled interest in deep learning by winning the ILSVRC challenge was *AlexNet*. This network was trained by extracting 224 × 224 × 3 patches from the images, which also defined the input sizes for the networks. The neural networks, which were entered into the ILSVRC contest in subsequent years, used a similar methodology of extracting patches.

Although most data augmentation methods are quite efficient, some forms of transformation that use principal component analysis (PCA) can be more expensive. PCA is used in order to change the color intensity of an image. If the computational costs are high, it becomes important to extract the images up front and store them. The basic idea here is to use the 3×3 covariance matrix of each pixel value and compute the principal components. Then, Gaussian noise is added to each principal component with zero mean and variance of 0.01. This noise is fixed over all the pixels of a particular image. The approach is dependent on the fact that object identity is invariant to color intensity and illumination. It is reported in [255] that data set augmentation reduces error rate by 1%.

One must be careful not to apply data augmentation blindly without regard to the data set and application at hand. For example, applying rotations and reflections on the MNIST data set [281] of handwritten digits is a bad idea because the digits in the data set are all presented in a similar orientation. Furthermore, the mirror image of an asymmetric digit is not a valid digit, and a rotation of a '6' is a '9.' The key point in deciding what types of data augmentation are reasonable is to account for the natural distribution of images in the full data set, as well as the effect of a specific type of data set augmentation on the class labels.

8.4 Case Studies of Convolutional Architectures

In the following, we provide some case studies of convolutional architectures. These case studies were derived from successful entries to the ILSVRC competition in recent years. These are instructive because they provide an understanding of the important factors in neural network design that can make these networks work well. Even though recent years have seen some changes in architectural design (like ReLU activation), it is striking how similar the modern architectures are to the basic design of *LeNet-5*. The main changes from *LeNet-5* to modern architectures are in terms of the explosion of depth, the use of ReLU activation, and the training efficiency enabled by modern hardware/optimization enhancements. Modern architectures are deeper, and they use a variety of computational, architectural, and hardware tricks to efficiently train these networks with large amounts of data. Hardware advancements should not be underestimated; modern GPU-based platforms are 10,000 times faster than the (similarly priced) systems available at the time *LeNet-5* was proposed. Even on these modern platforms, it often takes a week to train a convolutional neural network that is accurate enough to be competitive at ILSVRC. The hardware, data-centric, and algorithmic enhancements are connected to some extent. It is difficult to try new algorithmic tricks if enough data and computational power is not available to experiment with complex/deeper models in a reasonable amount of time. Therefore, the recent revolution in deep convolutional networks could not have been possible, had it not been for the large amounts of data and increased computational power available today.



(a) Without GPU partitioning

(b) With GPU partitioning (original architecture)

Figure 8.9: The *AlexNet* architecture. The ReLU activations follow each convolution layer, and are not explicitly shown. Note that the max-pooling layers are labeled as MP, and they follow only a subset of the convolution-ReLU combination layers. The architectural diagram in (b) is from [A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *NIPS Conference*, pp. 1097–1105. 2012.] ©2012 A. Krizhevsky, I. Sutskever, and G. Hinton.

In the following sections, we provide an overview of some of the well-known models that are often used for designing training algorithms for image classification. It is worth mentioning that some of these models are available as pretrained models over *ImageNet*, and the resulting features can be used for applications beyond classification. Such an approach is a form of transfer learning, which is discussed later in this section.

8.4.1 AlexNet

AlexNet was the winner of the 2012 ILSVRC competition. The architecture of *AlexNet* is shown in Figure 8.9(a). It is worth mentioning that there were two parallel pipelines of processing in the original architecture, which are not shown in Figure 8.9(a). These two pipelines are caused by two GPUs working together to build the training model with a faster speed and memory sharing. The network was originally trained on a GTX 580 GPU with 3 GB of memory, and it was impossible to fit the intermediate computations in this amount of space. Therefore, the network was partitioned across two GPUs. The original architecture is shown in Figure 8.9(b), in which the work is partitioned into two GPUs. We also show the architecture without the changes caused by the GPUs, so that it can be more easily compared with other convolutional neural network architectures discussed in this chapter. It is noteworthy that the GPUs are inter-connected in only a subset of the layers in Figure 8.9(b), which leads to some differences between Figure 8.9(a) and (b) in terms of the actual model constructed. Specifically, the GPU-partitioned architecture has fewer weights because not all layers have interconnections. Dropping some of the interconnections reduces the communication time between the processors and therefore helps in efficiency.

AlexNet starts with $224 \times 224 \times 3$ images and uses 96 filters of size $11 \times 11 \times 3$ in the first layer. A stride of 4 is used. This results in a first layer of size $55 \times 55 \times 96$. After the first layer has been computed, a max-pooling layer is used. This layer is denoted by 'MP' in Figure 8.9(a). Note that the architecture of Figure 8.9(a) is a simplified version of the architecture shown in Figure 8.9(b), which explicitly shows the two parallel pipelines. For example, Figure 8.9(b) shows a depth of the first convolution layer of only 48, because the 96 feature maps are divided among the GPUs for parallelization. On the other hand, Figure 8.9(a) does not assume the use of GPUs, and therefore the width is explicitly shown as 96. The ReLU activation function was applied after each convolutional layer, which was followed by response normalization and max-pooling. Although max-pooling has been annotated in the figure, it has not been assigned a block in the architecture. Furthermore, the ReLU and response normalization layers are not explicitly shown in the figure. These types of concise representations are common in pictorial depictions of neural architectures.

The second convolutional layer uses the response-normalized and pooled output of the first convolutional layer and filters it with 256 filters of size $5 \times 5 \times 96$. No intervening pooling or normalization layers are present in the third, fourth, or fifth convolutional layers. The sizes of the filters of the third, fourth, and fifth convolutional layers are $3 \times 3 \times 256$ (with 384 filters), $3 \times 3 \times 384$ (with 384 filters), and $3 \times 3 \times 384$ (with 256 filters). All max-pooling layers used 3×3 filters at stride 2. Therefore, there was some overlap among the pools. The fully connected layers have 4096 neurons. The final set of 4096 activations can be treated as a 4096-dimensional representation of the image. The final layer of *AlexNet* uses a 1000-way softmax in order to perform the classification. It is noteworthy that the final layer of 4096 activations (labeled by FC7 in Figure 8.9(b)) is often used to create a flat 4096 dimensional representation of an image for applications beyond classification. One can extract these features for any out-of-sample image by simply passing it through the trained neural network. These features often generalize well to other data sets and other tasks. Such features are referred to as FC7 features. In fact, the use of the extracted features from the penultimate layer as FC7 was popularized after *AlexNet*, even though the approach was known much earlier. As a result, such extracted features from the penultimate layer of a convolutional neural network are often referred to as *FC7 features*, irrespective of the number of layers in that network. It is noteworthy that the number of feature maps in middle layers is far larger than the initial depth of the volume in the input layer (which is only 3 corresponding to RGB colors) although their spatial dimensions are smaller. This is because the initial depth only contains the RGB color components, whereas the later layers capture different types of semantic features in the features maps.

Many design choices used in the architecture became standard in later architectures. A specific example is the use of ReLU activation in the architecture (instead of sigmoid or tanh units). The choice of the activation function in most convolutional neural networks today is almost exclusively focused on the ReLU, although this was not the case before *AlexNet*. Some other training tricks were known at the time, but their use in *AlexNet* popularized them. One example was the use of data augmentation, which turned out to be very useful in improving accuracy. *AlexNet* also underlined the importance of using specialized hardware like GPUs for training on such large data sets. Dropout was used with L_2-weight decay in order to improve generalization. The use of Dropout is common in virtually all types of architectures today because it provides an additional booster in most cases. The use of local response normalization was eventually discarded by later architectures.

We also briefly mention the parameter choices used in *AlexNet*. The interested reader can find the full code and parameter files of *AlexNet* at [584]. L_2-regularization was used with a parameter of 5×10^{-4}. *Dropout* was used by sampling units at a probability of 0.5. Momentum-based (mini-batch) stochastic gradient descent was used for training *AlexNet* with parameter value of 0.8. The batch-size was 128. The learning rate was 0.01, although it was eventually reduced a couple of times as the method began to converge. Even with the use of the GPU, the training time of *AlexNet* was of the order of a week.

The final top-5 error rate, which was defined as the fraction of cases in which the correct image was not included in the top-5 images, was about 15.4%. This error rate[3] was in comparison with the previous winners with an error rate of more than 25%. The gap with respect to the second-best performer in the contest was also similar. The use of single convolutional network provided a top-5 error rate of 18.2%, although using an ensemble of seven models provided the winning error-rate of 15.4%. Note that these types of ensemble-based tricks provide a consistent improvement of between 2% and 3% with most architectures. Furthermore, since the executions of most ensemble methods are embarrassingly parallelizable, it is relatively easy to perform them, as long as sufficient hardware resources are available. *AlexNet* is considered a fundamental advancement within the field of computer vision because of the large margin with which it won the ILSVRC contest. This success rekindled interest in deep learning in general, and convolutional neural networks in particular.

8.4.2 ZFNet

A variant of *ZFNet* [556] was the winner of the ILSVRC competition in 2013. Its architecture was heavily based on *AlexNet*, although some changes were made to further improve the accuracy. Most of these changes were associated with differences in hyperparameter choices, and therefore *ZFNet* is not very different from *AlexNet* at a fundamental level. One change from *AlexNet* to *ZFNet* was that the initial filters of size $11 \times 11 \times 3$ were changed to $7 \times 7 \times 3$. Instead of strides of 4, strides of 2 were used. The second layer used 5×5 filters at stride 2 as well. As in *AlexNet*, there are three max-pooling layers, and the same sizes of max-pooling filters were used. However, the first pair of max-pooling layers were performed after the first and second convolutions (rather than the second and third convolutions). As a result, the spatial footprint of the third layer changed to 13×13 rather than 27×27, although all other spatial footprints remained unchanged from *AlexNet*. The sizes of various layers in *AlexNet* and *ZFNet* are listed in Table 8.1.

The third, fourth, and fifth convolutional layers use a larger number of filters in *ZFNet* as compared to *AlexNet*. The number of filters in these layers were changed from $(384, 384, 256)$

[3] The top-5 error rate makes more sense in image data where a single image might contain objects of multiple classes. Throughout this chapter, we use the term “error rate” to refer to the top-5 error rate.

Table 8.1: Comparison of *AlexNet* and *ZFNet*

	AlexNet	*ZFNet*
Volume:	$224 \times 224 \times 3$	$224 \times 224 \times 3$
Operations:	Conv 11×11 (stride 4)	Conv 7×7 (stride 2), MP
Volume:	$55 \times 55 \times 96$	$55 \times 55 \times 96$
Operations:	Conv 5×5, MP	Conv 5×5 (stride 2), MP
Volume:	$27 \times 27 \times 256$	$13 \times 13 \times 256$
Operations:	Conv 3×3, MP	Conv 3×3
Volume:	$13 \times 13 \times 384$	$13 \times 13 \times 512$
Operations:	Conv 3×3	Conv 3×3
Volume:	$13 \times 13 \times 384$	$13 \times 13 \times 1024$
Operations:	Conv 3×3	Conv 3×3
Volume:	$13 \times 13 \times 256$	$13 \times 13 \times 512$
Operations:	MP, Fully connect	MP, Fully connect
FC6:	4096	4096
Operations:	Fully connect	Fully connect
FC7:	4096	4096
Operations:	Fully connect	Fully connect
FC8:	1000	1000
Operations:	Softmax	Softmax

to $(512, 1024, 512)$. As a result, the spatial footprints of *AlexNet* and *ZFNet* are the same in most layers, although the depths are different in the final three convolutional layers with similar spatial footprints. From an overall perspective, *ZFNet* used similar principles to *AlexNet*, and the main gains were obtained by changing the architectural parameters of *AlexNet*. This architecture reduced the top-5 error rate to 14.8% from 15.4%, and further increases in width/depth from the same author(s) reduced the error to 11.1%. Since most of the differences between *AlexNet* and *ZFNet* were those of minor design choices, this emphasizes the fact that small details are important when working with deep learning algorithms. Thus, extensive experimentation with neural architectures are sometimes important in order to obtain the best performance. The architecture of *ZfNet* was made wider and deeper, and the results were submitted to ILSVRC in 2013 under the name *Clarifai*, which was a company[4] founded by the first author of [556]. The difference[5] between *Clarifai* and *ZFNet* was one of width/depth of the network, although exact details of these differences are not available. This entry was the winning entry of the ILSVRC competition in 2013. Refer to [556] for details and a pictorial illustration of the architecture.

8.4.3 VGG

VGG [454] further emphasized the developing trend in terms of increased depth of networks. The tested networks were designed with various configurations with sizes between 11 and 19 layers, although the best-performing versions had 16 or more layers. *VGG* was a top-performing entry on ISLVRC in 2014, but it was not the winner. The winner was *GoogLeNet*, which had a top-5 error rate of 6.7% in comparison with the top-5 error rate of 7.3% for *VGG*. Nevertheless, *VGG* was important because it illustrated several important design principles that eventually became standard in future architectures.

[4] http://www.clarifai.com

[5] Personal communication from Matthew Zeiler.

An important innovation of *VGG* is that it reduced filter sizes but increased depth. It is important to understand that *reduced filter size necessitates increased depth.* This is because a small filter can capture only a small region of the image unless the network is deep. For example, a single feature that is a result of three sequential convolutions of size 3×3 will capture a region in the input of size 7×7. Note that using a single 7×7 filter directly on the input data will also capture the visual properties of a 7×7 input region. In the first case, we are using $3 \times 3 \times 3 = 27$ parameters, whereas we are using $7 \times 7 \times 1 = 49$ parameters in the second case. Therefore, the parameter footprint is smaller in the case when three sequential convolutions are used. However, three successive convolutions can often capture more interesting and complex features than a single convolution, and the resulting activations with a single convolution will look like primitive edge features. Therefore, the network with 7×7 filters will be unable to capture sophisticated shapes in smaller regions.

In general, greater depth forces more nonlinearity and greater regularization. A deeper network will have more nonlinearity because of the presence of more ReLU layers, and more regularization because the increased depth forces a structure on the layers through the use of repeated composition of convolutions. As discussed above, architectures with greater depth and reduced filter size require fewer parameters. This occurs in part because the number of parameters in each layer is given by the square of the filter size, whereas the number of parameters depend linearly on the depth. Therefore, one can drastically reduce the number of parameters by using smaller filter sizes, and instead "spend" these parameters by using increased depth. Increased depth also allows the use of a greater number of nonlinear activations, which increases the discriminative power of the model. Therefore *VGG* always uses filters with spatial footprint 3×3 and pooling of size 2×2. The convolution was done with stride 1, and a padding of 1 was used. The pooling was done at stride 2. Using a 3×3 filter with a padding of 1 maintains the spatial footprint of the output volume, although pooling always compresses the spatial footprint. Therefore, the pooling was done on non-overlapping spatial regions (unlike the previous two architectures), and always reduced the spatial footprint (i.e., both height and width) by a factor of 2. Another interesting design choice of *VGG* was that the number of filters was often increased by a factor of 2 after each max-pooling. The idea was to always increase the depth by a factor of 2 whenever the spatial footprint reduced by a factor of 2. This design choice results in some level of balance in the computational effort across layers, and was inherited by some of the later architectures like *ResNet*.

One issue with using deep configurations was that increased depth led to greater sensitivity with initialization, which is known to cause instability. This problem was solved by using pretraining, in which a shallower architecture was first trained, and then further layers were added. However, the pretraining was not done on a layer-by-layer basis. Rather, an 11-layer subset of the architecture was first trained. These trained layers were used to initialize a subset of the layers in the deeper architecture. *VGG* achieved a top-5 error of only 7.3% in the ISLVRC contest, which was one of the top performers but not the winner. The different configurations of *VGG* are shown in Table 8.2. Among these, the architecture denoted by column D was the winning architecture. Note that the number of filters increase by a factor of 2 after each max-pooling. Therefore, max-pooling causes the spatial height and width to reduce by a factor of 2, but this is compensated by increasing depth by a factor of 2. Performing convolutions with 3×3 filters and padding of 1 does not change the spatial footprint. Therefore, the sizes of each spatial dimension (i.e., height and width) in the regions between different max-pooling layers in column D of Table 8.2 are 224, 112, 56, 28, and 14, respectively. A final max-pooling is performed just before creating the fully connected layer, which reduces the spatial footprint further to 7. Therefore, the first fully

Table 8.2: Configurations used in *VGG*. The term C3D64 refers to the case in which convolutions are performed with 64 filters of spatial size 3×3 (and occasionally 1×1). The depth of the filter matches the corresponding layer. The padding of each filter is chosen in order to maintain the spatial footprint of the layer. All convolutions are followed by ReLU. The max-pool layer is referred to as M, and local response normalization as LRN. The softmax layer is denoted by S, and FC4096 refers to a fully connected layer with 4096 units. Other than the final set of layers, the number of filters always increases after each max-pooling. Therefore, reduced spatial footprint is often accompanied with increased depth.

Name:	**A**	**A-LRN**	**B**	**C**	**D**	**E**
# Layers	11	11	13	16	16	19
	C3D64	C3D64	C3D64	C3D64	C3D64	C3D64
		LRN	C3D64	C3D64	C3D64	C3D64
	M	M	M	M	M	M
	C3D128	C3D128	C3D128	C3D128	C3D128	C3D128
			C3D128	C3D128	C3D128	C3D128
	M	M	M	M	M	M
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
	C3D256	C3D256	C3D256	C3D256	C3D256	C3D256
				C1D256	C3D256	C3D256
						C3D256
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
	C3D512	C3D512	C3D512	C3D512	C3D512	C3D512
				C1D512	C3D512	C3D512
						C3D512
	M	M	M	M	M	M
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC4096	FC4096	FC4096	FC4096	FC4096	FC4096
	FC1000	FC1000	FC1000	FC1000	FC1000	FC1000
	S	S	S	S	S	S

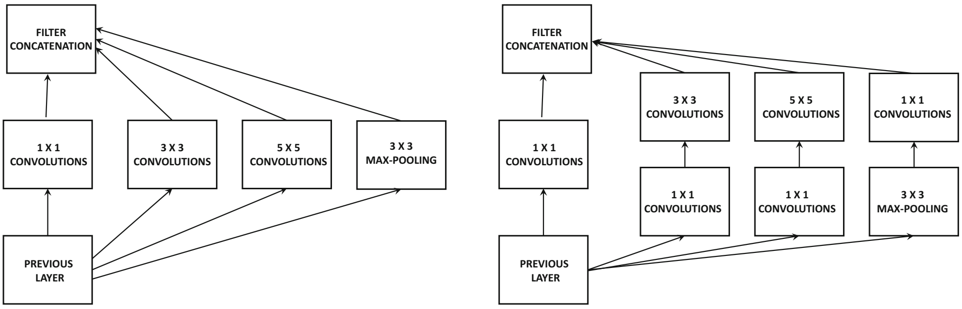
connected layer has dense connections between 4096 neurons and a $7 \times 7 \times 512$ volume. As we will see later, most of the parameters of the neural network are hidden in these connections.

An interesting exercise has been shown in [236] about where most of the parameters and the memory of the activations is located. In particular, the vast majority of the *memory* required for storing the activations and gradients in the forward and backward phases are required by the early part of the convolutional neural network with the largest spatial footprint. This point is significant because the memory required by a mini-batch is scaled by the size of the mini-batch. For example, it is has been shown in [236] that about 93MB are required for each image. Therefore, for a mini-batch size of 128, the total memory requirement would be about 12GB. Although the early layers require the most memory because of their large spatial footprints, they do not have a large parameter footprint because of the sparse connectivity and weight sharing. In fact, most of the parameters are required by the fully connected layers at the end. The connection of the final $7 \times 7 \times 512$ spatial layer (cf. column D in Table 8.2) to the 4096 neurons required $7 \times 7 \times 512 \times 4096 = 102,760,448$ parameters. The total number of parameters in *all* layers was about $138,000,000$. Therefore, *nearly* 75% *of the parameters are in a single layer of connections.* Furthermore, the majority of the remaining parameters are in the final two fully connected layers. In all, dense connectivity accounts for 90% of the parameter footprint in the neural network. This point is significant, as *GoogLeNet* uses some innovations to reduce the parameter footprint in the final layers.

It is notable that some of the architectures allow 1×1 convolutions. Although a 1×1 convolution does not combine the activations of spatially adjacent features, it does combine the feature values of different channels when the depth of a volume is greater than 1. Using a 1×1 convolution is also a way to incorporate additional nonlinearity into the architecture without making fundamental changes at the spatial level. This additional nonlinearity is incorporated via the ReLU activations attached to each layer. Refer to [454] for more details.

8.4.4 GoogLeNet

GoogLeNet proposed a novel concept referred to as an *inception architecture.* An inception architecture is a *network within a network.* The initial part of the architecture is much like a traditional convolutional network, and is referred to as the *stem.* The key part of the network is an intermediate layer, referred to as an *inception module.* An example of an inception module is illustrated in Figure 8.10(a). The basic idea of the inception module is that key information in the images is available at different levels of detail. If we use a large filter, we can capture information in a bigger area containing limited variation; if we use a smaller filter, we can capture detailed information in a smaller area. While one solution would be to pipe together many small filters, this would be wasteful of parameters and depth when it would suffice to use the broader patterns in a larger area. The problem is that we do not know up front which level of detail is appropriate for each region of the image. Why not give the neural network the flexibility to model the image at different levels of granularities? This is achieved with an inception module, which convolves with three different filter sizes in parallel. These filter sizes are 1×1, 3×3, and 5×5. A purely sequential piping of filters of the same size is inefficient when one is faced with objects of different scales in different images. Since all filters on the inception layer are learnable, the neural network can decide which ones will influence the output the most. By choosing filters of different sizes along different paths, different regions are represented at a different level of granularity. *GoogLeNet* is made up of nine inception modules that are arranged sequentially. Therefore, one can choose many alternative paths through the architecture, and the resulting features will represent very different spatial regions. For example, passing through four 3×3 filters



(a) Basic inception module (b) Implementation with 1 × 1 bottlenecks

Figure 8.10: The inception module of *GoogLeNet*

followed by only 1×1 filters will capture a relatively small spatial area. On the other hand, passing through many 5×5 filters will result in a much larger spatial footprint. In other words, the differences in the scales of the shapes captured in different hidden features will be magnified in later layers. In recent years, batch normalization has been used in conjunction with the inception architecture, which simplifies[6] the network structure from its original form.

One observation is that the inception module results in some computational inefficiency because of the large number of convolutions of different sizes. Therefore, an efficient implementation is shown in Figure 8.10(b), in which 1×1 convolutions are used to first reduce the depth of the feature map. This is because the number of 1×1 convolution filters is a modest factor less than the depth of the input volume. For example, one might first reduce an input depth of 256 to 64 by using 64 different 1×1 filters. These additional 1×1 convolutions are referred to as the *bottleneck operations* of the inception module. Initially reducing the depth of the feature map (with cheap 1×1 convolutions) saves computational efficiency with the larger convolutions because of the reduced depth of the layers after applying the bottleneck convolutions. One can view the 1×1 convolutions as a kind of supervised dimensionality reduction before applying the larger spatial filters. The dimensionality reduction is supervised because the parameters in the bottleneck filters are learned during backpropagation. The bottleneck also helps in reducing the depth after the pooling layer. The trick of bottleneck layers is also used in some other architectures, where it is helpful for improving efficiency and output depth.

The output layer of *GoogLeNet* also illustrates some interesting design principles. It is common to use fully connected layers near the output. However, *GoogLeNet* uses average pooling across the whole spatial area of the final set of activation maps to create a single value. Therefore, the number of features created in the final layer will be exactly equal to the number of filters. An important observation is that the vast majority of parameters are spent in connecting the final convolution layer to the first fully connected layer. This type of detailed connectivity is not required for applications in which only a class label needs to be predicted. Therefore, the average pooling approach is used. However, the average pooled representation completely loses all spatial information, and one must be careful of the types of applications it is used for. An important property of *GoogLeNet* was that it is extremely compact in terms of the number of parameters in comparison with *VGG*, and the number of

[6] The original architecture also contained auxiliary classifiers, which have been ignored in recent years.

parameters in the former is less by an order of magnitude. This is primarily because of the use of average pooling, which eventually became standard in many later architectures. On the other hand, the overall architecture of *GoogLeNet* is computationally more expensive.

The flexibility of *GoogLeNet* is inherent in the 22-layered inception architecture, in which objects of different scales are handled with the appropriate filter sizes. This flexibility of multigranular decomposition, which is enabled by the inception modules, was one of the keys to its performance. In addition, the replacement of the fully connected layer with average pooling greatly reduced the parameter footprint. This architecture was the winner of the ILSVRC contest in 2014, and *VGG* placed a close second. Even though *GoogLeNet* outperformed *VGG*, the latter does have the advantage of simplicity, which is sometimes appreciated by practitioners. Both architectures illustrated important design principles for convolution neural networks. The inception architecture has been the focus of significant research since then [486, 487], and numerous changes have been suggested to improve performance. In later years, a version of this architecture, referred to as *Inception-v4* [487], was combined with some of the ideas in *ResNet* (see next section) to create a 75-layer architecture with only 3.08% error.

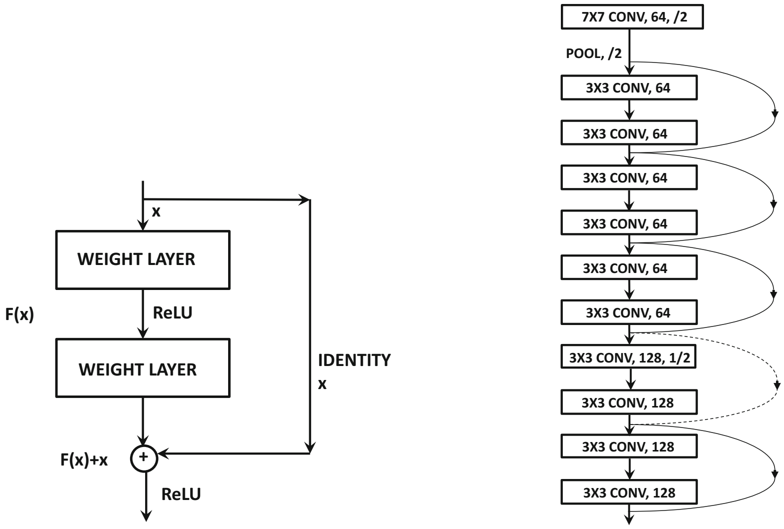
8.4.5 ResNet

ResNet [184] used 152 layers, which was almost an order of magnitude greater than previously used by other architectures. This architecture was the winner of the ILSVRC competition in 2015, and it achieved a top-5 error of 3.6%, which resulted in the first classifier with human-level performance. This accuracy is achieved by an ensemble of *ResNet* networks; even a single model achieves 4.5% accuracy. Training an architecture with 152 layers is generally not possible unless some important innovations are incorporated.

The main issue in training such deep networks is that the gradient flow between layers is impeded by the large number of operations in deep layers that can increase or decrease the size of the gradients. As discussed in Chapter 3, problems such as the vanishing and exploding gradients are caused by increased depth. However, the work in [184] suggests that the main training problem in such deep networks might not necessarily be caused by these problems, especially if batch normalization is used. The main problem is caused by the difficulty in getting the learning process to converge properly in a reasonable amount of time. Such convergence problems are common in networks with complex loss surfaces. Although some deep networks show large gaps between training and test error, the error on both the training ad test data is high in many deep networks. This implies that the optimization process has not made sufficient progress.

Although hierarchical feature engineering is the holy grail of learning with neural networks, its layer-wise implementations force all concepts in the image to require the same level of abstraction. Some concepts can be learned by using shallow networks, whereas others require fine-grained connections. For example, consider a circus elephant standing on a square frame. Some of the intricate features of the elephant might require a large number of layers to engineer, whereas the features of the square frame might require very few layers. Convergence will be unnecessarily slow when one is using a very deep network with a fixed depth across all paths to learn concepts, many of which can also be learned using shallow architectures. Why not let the neural network decide how many layers to use to learn each feature?

ResNet uses *skip connections* between layers in order to enable copying between layers and introduces an *iterative view* of feature engineering (as opposed to a hierarchical view). Long short-term memory networks and gated recurrent units leverage similar principles in sequence data by allowing portions of the states to be copied from one layer to the



(a) Skip-connections in residual module (b) Partial architecture of *ResNet*

Figure 8.11: The residual module and the first few layers of *ResNet*

next with the use of adjustable *gates*. In the case of *ResNet*, the non-existent "gates" are assumed to be always fully open. Most feed-forward networks only contain connections between layers i and $(i + 1)$, whereas *ResNet* contains connections between layers i and $(i + r)$ for $r > 1$. Examples of such skip connections, which form the basic unit of *ResNet*, are shown in Figure 8.11(a) with $r = 2$. This skip connection simply copies the input of layer i and adds it to the output of layer $(i+r)$. Such an approach enables effective gradient flow because the backpropagation algorithm now has a super-highway for propagating the gradients backwards using the skip connections. This basic unit is referred to as a *residual module*, and the entire network is created by putting together many of these basic modules. In most layers, an appropriately padded filter[7] is used with a stride of 1, so that the spatial size and depth of the input does not change from layer to layer. In such cases, it is easy to simply add the input of the ith layer to that of $(i + r)$. However, some layers do use strided convolutions to reduce each spatial dimension by a factor of 2. At the same time, depth is increased by a factor of 2 by using a larger number of filters. In such a case, one cannot use the identity function over the skip connection. Therefore, a linear projection matrix might need to be applied over the skip connection in order to adjust the dimensionality. This projection matrix defines a set of 1×1 convolution operations with stride of 2 in order to reduce spatial extent by factor of 2. The parameters of the projection matrix need to be learned during backpropagation.

In the original idea of *ResNet*, one only adds connections between layers i and $(i+r)$. For example, if we use $r = 2$, only skip connections only between successive odd layers are used. Later enhancements like *DenseNet* showed improved performance by adding connections between all pairs of layers. The basic unit of Figure 8.11(a) is repeated in *ResNet*, and therefore one can traverse the skip connections repeatedly in order to propagate input to the output after performing very few forward computations. An example of the first few

[7] Typically, a 3×3 filter is used at a stride/padding of 1. This trend started with the principles in *VGG*, and was adopted by *ResNet*.

layers of the architecture is shown in Figure 8.11(b). This particular snapshot is based on the first few layers of the 34-layer architecture. Most of the skip connections are shown in solid lines in Figure 8.11(b), which corresponds to the use of the identity function with an unchanged filter volume. However, in some layers, a stride of 2 is used, which causes the spatial and depth footprint to change. In these layers, a projection matrix needs to be used, which is denoted by a dashed skip connection. Four different architectures were tested in the original work [184], which contained 34, 50, 101, and 152 layers, respectively. The 152-layer architecture had the best performance, but even the 34-layer architecture performed better than did the best-performing ILSVRC entry from the previous year.

The use of skip connections provides paths of unimpeded gradient flow and therefore has important consequences for the behavior of the backpropagation algorithm. The skip connections take on the function of super-highways in enabling gradient flow, creating a situation where multiple paths of variable lengths exist from the input to the output. In such cases, the shortest paths enable the most learning, and the longer paths can be viewed as residual contributions. This gives the learning algorithm the flexibility of choosing the appropriate level of nonlinearity for a particular input. Inputs that can be classified with a small amount of nonlinearity will skip many connections. Other inputs with a more complex structure might traverse a larger number of connections in order to extract the relevant features. Therefore, the approach is also referred to as residual learning, in which learning along longer paths is a kind of fine tuning of the easier learning along shorter paths. In other words, the approach is well suited to cases in which different aspects of the image have different levels of complexity. The work in [184] shows that the residual responses from deeper layers are often relatively small, which validates the intuition that fixed depth is an impediment to proper learning. In such cases, the convergence is often not a problem, because the shorter paths enable a significant portion of the learning with unimpeded gradient flows. An interesting insight in [505] is that *ResNet* behaves like an ensemble of shallow networks because many alternative paths of shorter length are enabled by this type of architecture. Only a small amount of learning is enabled by the deeper paths, and only when it is absolutely necessary. The work in [505] in fact provides a pictorial depiction of an unraveled architecture of *ResNet* in which the different paths are explicitly shown in a parallel pipeline. This unraveled view provides a clear understanding of why *ResNet* has some similarities with ensemble-centric design principles. A consequence of this point of view is that dropping some of the layers from a trained *ResNet* at prediction time does not degrade accuracy as significantly as other networks like *VGG*.

More insights can be obtained by reading the work on *wide residual networks* [549]. This work suggests that increased depth of the residual network does not always help because most of the extremely deep paths are not used anyway. The skip connections do result in alternative paths and effectively increase the width of the network. The work in [549] suggests that better results can be obtained by limiting the total number of layers to some extent (say, 50 instead of 150), and using an increased number of filters in each layer. Note that a depth of 50 is still quite large from pre-*ResNet* standards, but is low compared to the depth used in recent experiments with residual networks. This approach also helps in parallelizing operations.

Variations of Skip Architectures

Since the architecture of *ResNet* was proposed, several variations were suggested to further improve performance. For example, the independently proposed *highway networks* [161]

Table 8.3: The number of layers in various top-performing ILSVRC contest entries

Name	Year	Number of Layers	Top-5 Error
–	Before 2012	≤ 5	$> 25\%$
AlexNet	2012	8	15.4%
ZfNet/Clarifai	2013	$8/> 8$	14.8% / 11.1%
VGG	2014	19	7.3%
GoogLeNet	2014	22	6.7%
ResNet	2015	152	3.6%

introduced the notion of gated skip connections, and can be considered a more general architecture. In highway networks, gates are used in lieu of the identity mapping, although a closed gate does not pass a lot of information through. In such cases, gating networks do not behave like residual networks. However, residual networks can be considered special cases of gating networks in which the gates are always fully open. Highway networks are closely related to both LSTMs and *ResNets*, although *ResNets* still seem to perform better in the image recognition task because of their focus on enabling gradient flow with multiple paths. The original *ResNet* architecture uses a simple block of layers between skip connections. However, the *ResNext* architecture varies on this principle by using inception modules between skip connections [537].

Instead of using skip connections, one can use convolution transformations between every pair of layers [211]. Therefore, instead of the L transformations in a feed-forward network with L layers, one is using $L(L-1)/2$ transformations. In other words, the concatenation of all the feature maps of the previous $(l-1)$ layers is used by the lth layer. This architecture is referred to as *DenseNet*. Note that the goal of such an architecture is similar to that of skip connections by allowing each layer to learn from whatever level of abstraction is useful.

An interesting variant that seems to work well is the use of *stochastic depth* [210] in which some of the blocks between skip connections are randomly dropped during training time, but the full network is used during testing time. Note that this approach seems similar to *Dropout*, which makes the network thinner rather than shallower by dropping nodes. However, *Dropout* has somewhat different motivations from layer-wise node dropping, because the latter is more focused on improving gradient flow rather than preventing feature co-adaptation.

8.4.6 The Effects of Depth

The significant advancements in performance in recent years in the ILSVRC contest are mostly a result of improved computational power, greater data availability, and changes in architectural design that have enabled the effective training of neural networks with increased depth. These three aspects also support each other, because experimentation with better architectures is only possible with sufficient data and improved computational efficiency. This is also one of the reasons why the fine-tuning and tweaks of relatively old architectures (like recurrent neural networks) with known problems were not performed until recently.

The number of layers and the error rates of various networks are shown in Table 8.3. The rapid increase in accuracy in the short period from 2012 to 2015 is quite remarkable, and is unusual for most machine learning applications that are as well studied as image recognition. Another important observation is that increased depth of the neural network is closely correlated with improved error rates. Therefore, an important focus of the research in recent years has been to enable algorithmic modifications that support increased depth of the neural architecture. It is noteworthy that convolutional neural networks are among the deepest of all classes of neural networks. Interestingly, traditional feed-forward networks in other domains do not need to be very deep for most applications like classification. Indeed, the coining of the term "deep learning" owes a lot of its origins to the impressive performances of convolutional neural networks and specific improvements observed with increased depth.

8.4.7 Pretrained Models

One of the challenges faced by analysts in the image domain is that *labeled* training data may not even be available for a particular application. Consider the case in which one has a set of images that need to be used for image retrieval. In retrieval applications, labels are not available but it is important for the features to be semantically coherent. In some other cases, one might wish to perform classification on a data set with a particular set of labels, which might be limited in availability and different from large resources like *ImageNet*. These settings cause problems because neural networks require a lot of training data to build from scratch.

However, a key point about image data is that the extracted features from a particular data set are highly reusable across data sources. For example, the way in which a cat is represented will not vary a lot if the same number of pixels and color channels are used in different data sources. In such cases, generic data sources, which are representative of a wide spectrum of images, are useful. For example, the *ImageNet* data set [581] contains more than a million images drawn from 1000 categories encountered in everyday life. The chosen 1000 categories and the large diversity of images in the data set are representative and exhaustive enough that one can use them to extract features of images for general-purpose settings. For example, the features extracted from the *ImageNet* data can be used to represent a completely different image data set by passing it through a pretrained convolutional neural network (like *AlexNet*) and extracting the multidimensional features from the fully connected layers. This new representation can be used for a completely different application like clustering or retrieval. This type of approach is so common, that *one rarely trains convolutional neural networks from scratch*. The extracted features from the penultimate layer are often referred to as FC7 features, which is an inheritance from the name of the layer in *AlexNet*. Of course, an arbitrary convolutional network might not have the same number of layers as *AlexNet*; however, the name FC7 has stuck.

This type of off-the-shelf feature extraction approach [390] can be viewed as a kind of *transfer learning*, because we are using a public resource like *ImageNet* to extract features to solve different problems in settings where enough training data is not available. Such an approach has become standard practice in many image recognition tasks, and many software frameworks like *Caffe* provide ready access to these features [585, 586]. In fact, *Caffe* provides a "zoo" of such pretrained models, which can be downloaded and used [586]. If some additional training data is available, one can use it to fine-tune only the deeper layers (i.e., layers closer to the output layer). The weights of the early layers (closer to the input) are fixed. The reason for training only the deeper layers, while keeping the early layers fixed, is that the earlier layers capture only primitive features like edges, whereas the deeper layers capture more complex features. The primitive features do not change too much with the application at hand, whereas the deeper features might be sensitive to the application at hand. For example, all types of images will require edges of different orientation to represent them (captured in early layers), but a feature corresponding to the wheel of a truck will be relevant to a data set containing images of trucks. In other words, early layers tend to capture highly generalizable features (across different computer vision data sets), whereas later layers tend to capture data-specific features. A discussion of the transferability of features derived from convolutional neural networks across data sets and tasks is provided in [361].

8.5 Visualization and Unsupervised Learning

An interesting property of convolutional neural networks is that they are highly interpretable in terms of the types of features they can learn. However, it takes some effort to actually interpret these features. The first approach that comes to mind is to simply visualize the 2-dimensional (spatial) components of the filters. Although this type of visualization can provide some interesting visualizations of the primitive edges and lines learned in the first layer of the neural network, it is not very useful for later layers. In the first layer, it is possible to visualize these filters because they operate directly on the input image, and often tend to look like primitive parts of the image (such as edges). However, it is not quite as simple a matter to visualize these filters in later layers because they operate on input volumes that have already been scrambled with convolution operations. In order to obtain any kind of interpretability one must find a way to map the impacts of all operations all the way back to the input layer. Therefore, the goal of visualization is often to identify and highlight the portions of the input image to which a particular hidden feature is responding. For example, the value of one hidden feature might be sensitive to changes in the portion of the image corresponding to the wheel of a truck, and a different hidden feature might be sensitive to its hood. This is naturally achieved by computing the sensitivity (i.e., gradient) of a hidden feature with respect to each pixel of the input image. As we will see, these types of visualizations are closely related to backpropagation, unsupervised learning, and transposed convolutional operations (used for creating the decoder portions of autoencoders). Therefore, this chapter will discuss these closely related topics in an integrated way.

There are two primary settings in which one can encode and decode an image. In the first setting, the compressed feature maps are learned by using any of the supervised models discussed in earlier sections. Once the network has been trained in a supervised way, one can attempt to reconstruct the portions of the image that most activate a given feature. Furthermore, the portions of an image that are most likely to activate a particular hidden feature or a class are identified. As we will see later, this goal can be achieved with various types of backpropagation and optimization formulations. The second setting is purely unsupervised, in which a convolutional network (encoder) is hooked up to a deconvolutional network (decoder). As we will see later, the latter is also a form of transposed convolution, which is similar to backpropagation. However, in this case, the weights of the encoder and decoder are learned jointly to minimize the reconstruction error. The first setting is obviously simpler because the encoder is trained in a supervised way, and one only has to learn the effect of different portions of the input field on various hidden features. In the second setting, the entire training and learning of weights of the network has to be done from scratch.

8.5.1 Visualizing the Features of a Trained Network

Consider a neural network that has already been trained using a large data set like *ImageNet*. The goal is to visualize and understand the impact of the different portions of the input image (i.e., receptive field) on various features in the hidden layers and the output layer (e.g., the 1000 softmax outputs in *AlexNet*). We would like to answer the following questions:

1. Given an activation of a feature anywhere in the neural network for a *particular* input image, visualize the portions of the input to which that feature is responding the most. Note that the feature might be one of the hidden features in the spatially arranged layers, in the fully connected hidden layers (e.g., FC7), or even one of the softmax outputs. In the last of these cases, one obtains some insight of the specific relationship of a particular input image to a class. For example, if an input image is activating the label for "*banana*," we hope to see the parts of the specific input image that look most like a banana.

2. Given a particular feature anywhere in the neural network, find a fantasy image that is likely to activate that feature the most. As in the previous case, the feature might be one of the hidden features or even one of the features from the softmax outputs. For example, one might want to know what type of fantasy image is most likely to classify to a "*banana*" in the trained network at hand.

In both these cases, the easiest approach to visualize the impact of specific features is to use gradient-based methods. The second of the above goals is rather hard, and one often does not obtain satisfactory visualizations without careful regularization.

Gradient-Based Visualization of Activated Features

The backpropagation algorithm that is used to train the neural network is also helpful for gradient-based visualization. It is noteworthy that backpropagation-based gradient computation is a form of transposed convolution. In traditional autoencoders, transposed weight matrices (of those used in the encoder layer) are often used in the decoder. Therefore, the connections between backpropagation and feature reconstruction are deep and are applicable across all types of neural networks. The main difference from the traditional backpropagation setting is that our end-goal is to determine the sensitivity of the hidden/output features with respect to *different pixels of the input image* rather than with respect to the weights. However, even traditional backpropagation does compute the sensitivity of the outputs with respect to various layers as an intermediate step, and therefore almost exactly the same approach can be used in both cases.

When the sensitivity of an output o is computed with respect to the input pixels, the visualization of this sensitivity over the corresponding pixels is referred to as a *saliency map* [456]. For example, the output o might be the softmax probability (or unnormalized score before applying softmax) of the class "*banana*." Then, for each pixel x_i in the image, we would like to determine the value of $\frac{\partial o}{\partial x_i}$. This value can be computed by straightforward backpropagation all the way[8] to the input layer. The softmax probability of "*banana*" will be relatively insensitive to small changes in those portions of the image that are irrelevant to the recognition of a banana. Therefore, the values of $\frac{\partial o}{\partial x_i}$ will be close to 0 for such

[8] Under normal circumstances, one only backpropagates to hidden layers as an intermediate step to compute gradients with respect to incoming weights in that hidden layer. Therefore, backpropagation to input layer is never really needed in traditional training. However, backpropagation to the input layer is identical to that with respect to the hidden layers.

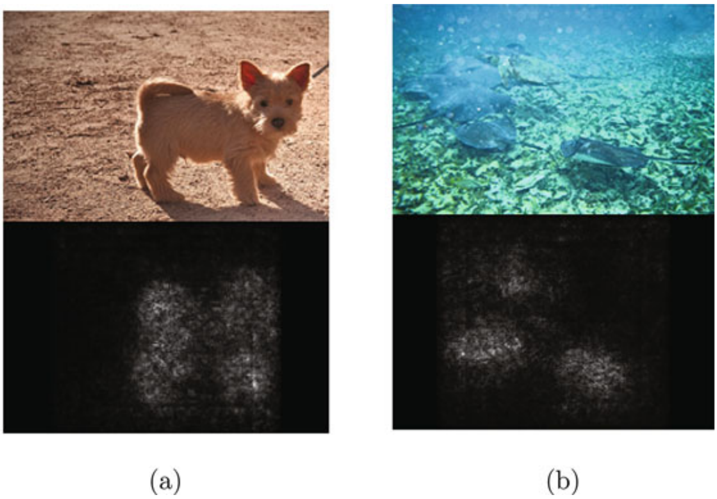


Figure 8.12: Examples of portions of specific images activated by particular class labels. These images appear in the work by Simonyan, Vedaldi, and Zisserman [456]. Reproduced with permission. (©2014 Simonyan, Vedaldi, and Zisserman)

irrelevant regions, whereas the portions of the image that define a banana will have large magnitudes. For example, in the case of *AlexNet*, the entire $224 \times 224 \times 3$ volume defined by $\frac{\partial o}{\partial x_i}$ of *backpropagated gradients* will have portions with large magnitudes corresponding to the banana in the image. To visualize this volume, we first convert it to grayscale by taking the *maximum of the absolute magnitude of the gradient* over the three RGB channels to create a $224 \times 224 \times 1$ map with only non-negative values. The bright portions of this grayscale visualization will tell us which portion of the input image are relevant to the banana. Examples of grayscale visualizations of the portions of the image that excite relevant classes are shown in Figure 8.12. For example, the bright portion of the image in Figure 8.12(a) excites the animal in the image, which also represents its class label. As discussed in Section 2.4 of Chapter 2, this type of approach can also be used for interpretability and feature selection in traditional neural networks (and not just convolutional methods).

This general approach has also been used for visualizing the activations of specific hidden features. Consider the value h of a hidden variable for a particular input image. How is this variable responding to the input image at its current activation level? The idea is that if we slightly increase or decrease the color intensity of some pixels, the value of h will be affected more than if we increase or decrease other pixels. First, the hidden variable h will be affected by a small rectangular portion of the image (i.e., receptive field), which is very small when h is present in early layers but much larger in later layers. For example, the receptive field of h might only be of size 3×3 when it is selected from the first hidden layer in the case of *VGG*. Examples of the image crops corresponding to specific images in which a particular neuron in a hidden layer is highly activated are shown in each row on the right-hand side of Figure 8.13. Note that each row contains a somewhat similar image. This is not a coincidence because that row corresponds to a particular hidden feature, and the variations in that row are caused by the different choices of image. Note that the choices of the image for a row is also not random, because we are selecting the images that most activate that feature. Therefore, all the images will contain the same visual characteristic that cause this hidden feature to be activated. The grayscale portion of the visualization corresponds to the sensitivity of the feature to the pixel-specific values in the corresponding image crop.

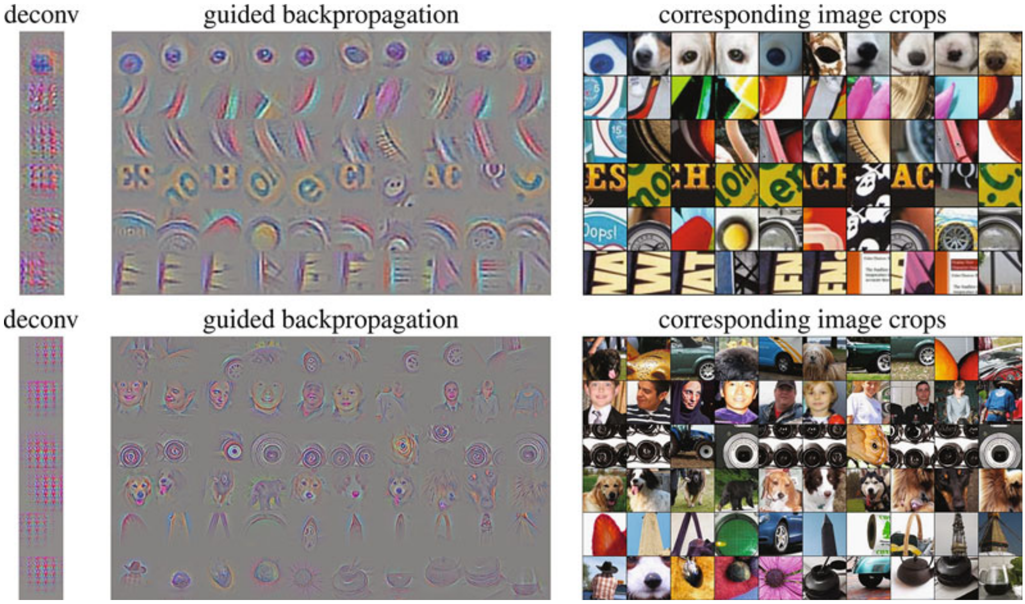
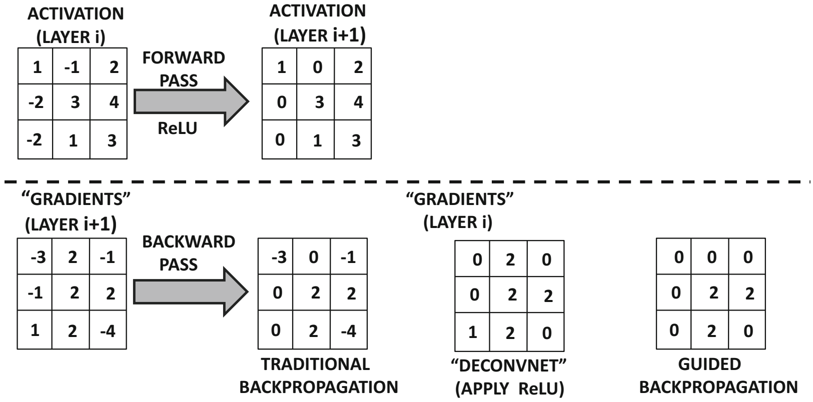


Figure 8.13: Examples of activation visualizations in different layers in Springenberg *et al.*'s work [466]. Reprinted from [466] with permission (©2015 Springenberg, Dosovitskiy, Brox, Riedmiller).

At a high level of activation level of h, some of the pixels in that receptive field will be more sensitive to h than others. By isolating the pixels to which the hidden variable h has the greatest sensitivity and visualizing the corresponding rectangular regions, one can get an idea of what part of the input map most affects a particular hidden feature. Therefore, any particular pixel x_i, we want to compute $\frac{\partial h}{\partial x_i}$, and then visualize those pixels with large values of this gradient. However, instead of backpropagation, the notions of *"deconvnet"* [556] and *guided backpropagation* [466] are sometimes used. The notion of "deconvent" is also used in convolutional autoencoders. The main difference is in terms of how the gradient of the ReLU nonlinearity is propagated backwards. As discussed in Table 3.1 of Chapter 3, the partial derivative of a ReLU unit is copied backwards during backpropagation if the *input* to the ReLU is positive, and is otherwise set to 0. However, in "deconvnet," the partial derivative of a ReLU unit is copied backwards, if this partial derivative is itself larger than 0. This is like using a ReLU on the propagated gradient in the backward pass. In other words, we replace $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0)$ in Table 3.1 with $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{g}_{i+1} > 0)$. Here $\overline{z}_i$ represents the forward activations, and $\overline{g}_i$ represents the backpropagated gradients with respect to the ith layer containing only ReLU units. The function $I(\cdot)$ is an element-wise indicator function, which takes on the value of 1 for each element in the vector argument when the condition is true for that element. In guided backpropagation, we *combine* the conditions used in traditional backpropagation and ReLU by using $\overline{g}_i = \overline{g}_{i+1} \odot I(\overline{z}_i > 0) \odot I(\overline{g}_{i+1} > 0)$. A pictorial illustration of the three variations of backpropagation is shown in Figure 8.14. It is suggested in [466] that guided backpropagation gives better visualizations than "deconvnet," which in turn gives better results than traditional backpropagation.



ACTIVATION (LAYER i)

1	-1	2
-2	3	4
-2	1	3

ACTIVATION (LAYER i+1)

1	0	2
0	3	4
0	1	3

"GRADIENTS" (LAYER i+1)

-3	2	-1
-1	2	2
1	2	-4

TRADITIONAL BACKPROPAGATION

-3	0	-1
0	2	2
0	2	-4

"DECONVNET" (APPLY ReLU)

0	2	0
0	2	2
1	2	0

GUIDED BACKPROPAGATION

0	0	0
0	2	2
0	2	0

Figure 8.14: The different variations of backpropagation of ReLU for visualization

One way of interpreting the difference between traditional backpropagation and "deconvnet" is by interpreting backwards propagation of gradients as the operations of a decoder with transposed convolutions with respect to the encoder [456]. However, in this decoder, we are again using the ReLU function rather than the gradient-based transformation implied by the ReLU. After all, all forms of decoders use the same activation functions as the encoder. Another feature of the visualization approach in [466] is that it omits the use of pooling layers in the convolutional neural network altogether, and instead relies on strided convolutions. The work in [466] identified several highly activated neurons in specific layers corresponding to specific input images and provided visualizations of the rectangular regions of those images corresponding to the receptive fields of those hidden neurons. We have already discussed earlier that the right-hand side of Figure 8.13 contains the input regions corresponding to specific neurons in hidden layers. The left-hand side of Figure 8.13 also shows the specific characteristics of each image that excite that particular neuron. The visualization on the left-hand side is obtained with guided backpropagation. Note that the upper set of images correspond to the sixth layer, whereas the lower set of images corresponds to the ninth layer of the convolutional network. As a result, the images in the lower set typically corresponds to larger regions of the input image containing more complex shapes.

Another excellent set of visualizations from [556] is shown in Figure 8.15. The main difference is that the work in [556] also uses max-pooling layers, and is based on deconvolutions rather than guided backpropagation. The specific hidden variables chosen are the top-9 largest activations in each feature map. In each case, the relevant square region of the image is shown together with the corresponding visualization. It is evident that the hidden features in early layers correspond to primitive lines, which become increasingly more complex in later layers. This is one of the reasons that convolutional neural networks are viewed as methods that create hierarchical features. The features in early layers tend to be more generic, and they can be used across a wider variety of data sets. The features in later layers tend to be more specific to individual data sets. This is a key property exploited in transfer learning applications, in which pretrained networks are broadly used, and only the later layers are fine-tuned in a manner that's specific to data set and application.

Synthesized Images that Activate a Feature

The above examples tell us the portions of a *particular* image that most affect a particular neuron. A more general question is to ask what kind of image patch would maximally activate a particular neuron. For ease in discussion, we will discuss the case in which the neuron is an output value o of a particular class (i.e., unnormalized output before applying softmax). For example, the value of o might be the unnormalized score for "*banana*." Note that one can also apply a similar approach to intermediate neurons rather than the class score. We would like to learn the input image $\overline{x}$ that maximizes the output o, while applying regularization to $\overline{x}$:

$$\text{Maximize}_{\overline{x}}\ J(\overline{x}) = (o - \lambda||\overline{x}||^2)$$

Here, λ is the regularization parameter, and is important in order to extract semantically interpretable images. One can use gradient ascent in conjunction with backpropagation in order to learn the input image $\overline{x}$ that maximizes the above objective function. Therefore, we start with a zero image $\overline{x}$ and update $\overline{x}$ using gradient ascent in conjunction with backpropagation with respect to the above objective function. In other words, the following update is used:

$$\overline{x} \Leftarrow \overline{x} + \alpha \nabla_{\overline{x}} J(\overline{x}) \tag{8.4}$$

Here, α is the learning rate. The key point is that backpropagation is being leveraged in an unusual way to update the *image pixels* while keeping the (already learned) weights fixed. Examples of synthesized images for three classes are shown in Figure 8.16. Other advanced methods for generating more realistic images on the basis of class labels are discussed in [358].

8.5.2 Convolutional Autoencoders

The use of the autoencoder in traditional neural networks is discussed in Chapters 2 and 4. Recall that the autoencoder reconstructs data points after passing them through a compression phase. In some cases, the data is not compressed although the representations are sparse. The portion before the most compressed layer of the architecture is referred to as the encoder, and the portion after the compressed portion is referred to as the decoder. We repeat the pictorial view of the encoder-decoder architecture for the traditional case in Figure 8.17(a). The convolutional autoencoder has a similar principle, which reconstructs images after passing them through a compression phase. The main difference between a traditional autoencoder and a convolutional autoencoder is that the latter is focused on using spatial relationships between points in order to extract features that have a visual interpretation. The spatial convolution operations in the intermediate layers achieve precisely this goal. An illustration of the convolutional autoencoder is shown in Figure 8.17(b) in comparison with the traditional autoencoder in Figure 8.17(a). Note the 3-dimensional spatial shape of the encoder and decoder in the second case. However, it is possible to conceive of several variations to this basic architecture. For example, the codes in the middle can either be spatial or they can be flattened with the use of fully connected layers, depending on the application at hand. The fully connected layers would be necessary to create a multidimensional code that can be used with arbitrary applications (without worrying about spatial constraints among features). In the following, we will simplify the discussion by assuming that the compressed code in the middle is spatial in nature.

Just as the compression portion of the encoder uses a convolution operation, the decompression operation uses a *deconvolution* operation. Similarly, pooling is matched with an

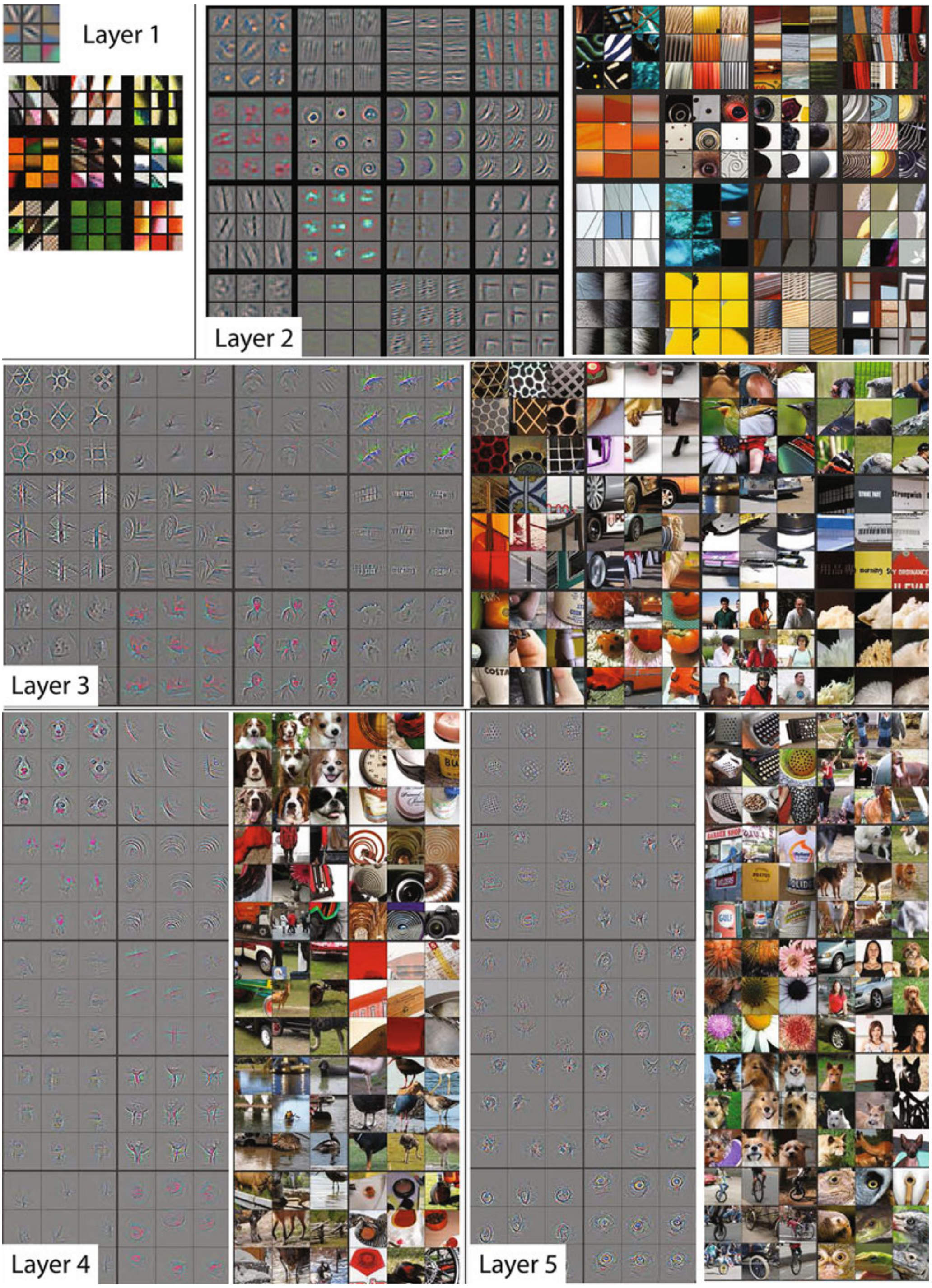
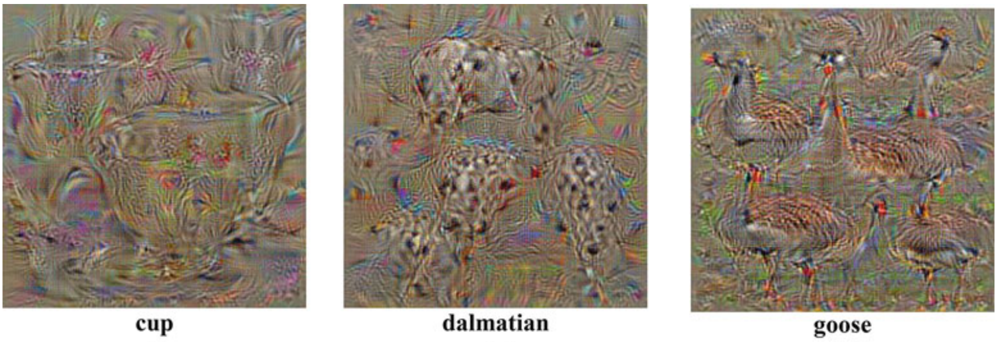
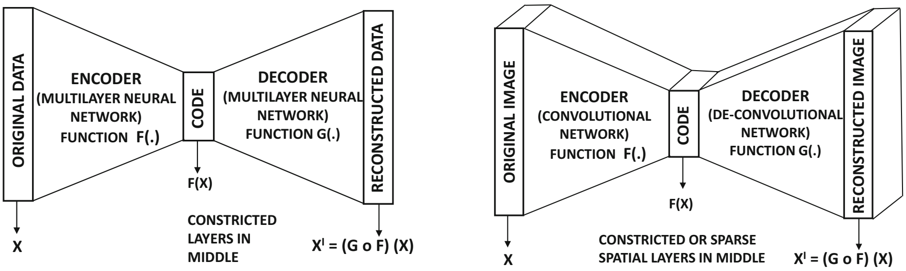


Figure 8.15: Examples of activation visualizations in different layers based on Zeiler and Fergus's work [556]. Reprinted from [556] with permission. ©Springer International Publishing Switzerland, 2014.



(a) (b) (c)

Figure 8.16: Examples of synthesized images with respect to particular class labels. These examples appear in the work by Simonyan, Vedaldi, and Zisserman [456]. Reproduced with permission (©2014 Simonyan, Vedaldi, and Zisserman)


ORIGINAL DATA
ENCODER (MULTILAYER NEURAL NETWORK) FUNCTION F(.)
CODE
DECODER (MULTILAYER NEURAL NETWORK) FUNCTION G(.)
RECONSTRUCTED DATA
X
F(X)
CONSTRICTED LAYERS IN MIDDLE
X' = (G o F) (X)
ORIGINAL IMAGE
ENCODER (CONVOLUTIONAL NETWORK) FUNCTION F(.)
CODE
DECODER (DE-CONVOLUTIONAL NETWORK) FUNCTION G(.)
RECONSTRUCTED IMAGE
X
F(X)
CONSTRICTED OR SPARSE SPATIAL LAYERS IN MIDDLE
X' = (G o F) (X)

(a) A traditional autoencoder architecture (b) Convolutional autoencoder

Figure 8.17: A traditional autoencoder and a convolutional autoencoder.

unpooling operation. Deconvolution is also referred to as *transposed convolution.* Interestingly, the transposed convolution operation is the same as that used for backpropagation. The term "deconvolution" is perhaps a bit misleading because every deconvolution is in actuality a convolution with a filter that is derived by transposing and inverting the tensor representing the original convolution filter (cf. Figure 8.7 and Equation 8.3). We can already see that deconvolution uses similar principles to that of backpropagation. The main difference is in terms of how the ReLU function is handled, which makes deconvolution more similar to "deconvnet" or *guided* backpropagation. In fact, the decoder in a convolutional autoencoder performs similar operations to the backpropagation phase of gradient-based visualization. Some architectures do away with the pooling and unpooling operations, and work with only convolution operations (together with activation functions). A notable example is the design of *fully convolutional networks* [449, 466].

The fact that the deconvolution operation is really not much different from a convolution operation is not surprising. Even in traditional feed-forward networks, the decoder part of the network performs the same types of matrix multiplications as the encoder part of the network, except that the transposed weight matrices are used. One can summarize the analogy between traditional autoencoders and convolutional autoencoders in Table 8.4. Note that the relationship between forward backpropagation and backward propagation is

Table 8.4: The relationship between backpropagation and decoders

Linear Operation	**Traditional neural networks**	**Convolutional neural networks**
Forward Propagation	Matrix multiplication	Convolution
Backpropagation	Transposed matrix multiplication	Transposed convolution
Decoder layer	Transposed matrix multiplication (Identical to backpropagation)	Transposed convolution (Identical to backpropagation)

similar in traditional and convolutional neural networks in terms of how the corresponding matrix operations are performed. A similar observation is true about the nature of the relationship between encoders and decoders.

There are three operations corresponding to the convolution, max-pooling, and the ReLU nonlinearity. The goal is to perform the inversion of the operations in the decoder layer that have been performed in the encoder layer. There is no easy way to exactly invert some of the operations (such as max-pooling and ReLU). However, excellent image reconstruction can still be achieved with the proper design choices. First, we describe the case of an autoencoder with a single layer with convolution, ReLU, and max-pooling. Then, we discuss how to generalize it to multiple layers.

Although one typically wants to use the inverse of the encoder operations in the decoder, the ReLU is not an invertible function because a value of 0 has many possible inversions. Therefore, a ReLU is replaced by another ReLU in the decoder layer (although other options are possible). Therefore, the architecture of this simple autoencoder is as follows:

$$\underbrace{\text{Convolve} \Rightarrow \text{ReLU} \Rightarrow \text{Max-Pool}}_{\text{Encoder}} \Rightarrow \text{Code} \Rightarrow \underbrace{\text{Unpool} \Rightarrow \text{ReLU} \Rightarrow \text{De-Convolve}}_{\text{Decoder}}$$

Note that the layers are symmetrically arranged in terms of how a matching layer in the decoder undoes the effect of a corresponding layer in the encoder. However, there are many variations to this basic theme. For example, the ReLU might be placed after the deconvolution. Furthermore, in some variations [310], it is recommended to use deeper encoders than the decoders with an asymmetric architecture. However, with a stacked variation of the symmetric architecture above, it is possible to train just the encoder with a classification output layer (and a supervised data set like *ImageNet*) and then use its symmetric decoder (with transposed/inverted filters) to perform "deconvnet" visualization [556]. Although one can always use this approach to initialize the autoencoder, we will discuss enhancements of this concept where the encoder and decoder are jointly trained in an unsupervised way.

We will count each layer like convolution and ReLU as a separate layer here, and therefore we have a total of seven layers including the input. This architecture is simplistic because it uses a single convolution layer in each of the encoders and decoders. In more generalized architectures, these layers are stacked to create more powerful architectures. However, it is helpful to illustrate the relationship of the basic operations like unpooling and deconvolution to their encoding counterparts (like pooling and convolution). Another simplification is that the code is contained in a spatial layer, whereas one could also insert fully connected layers in the middle. Although this example (and Figure 8.17(b)) uses a spatial code, the use of fully connected layers in the middle is more useful for practical applications. On the other hand, the spatial layers in the middle can be used for visualization.

Consider a situation in which the encoder uses d_2 square filters of size $F_1 \times F_1 \times d_1$ in the first layer. Also assume that the first layer is a (spatially) square volume of size $L_1 \times L_1 \times d_1$.

The (i, j, k)th entry of the pth filter in the first layer has weight $w_{ijk}^{(p,1)}$. This notations are consistent with those used in Section 8.2, where the convolution operation is defined. It is common to use the precise level of padding required in the convolution layer, so that the feature maps in the second layer are also of size L_1. This level of padding is $F_1 - 1$, which is referred to as *half-padding*. However, it is also possible to use no padding in the convolution layer, if one uses full padding in the corresponding deconvolution layer. In general, the sum of the paddings between the convolution and its corresponding deconvolution layer must sum to $F_1 - 1$ in order to maintain the spatial size of the layer in a convolution-deconvolution pair.

Here, it is important to understand that although each $W^{(p,1)} = [w_{ijk}^{(p,1)}]$ is a 3-dimensional tensor, one can create a 4-dimensional tensor by including the index p in the tensor. The deconvolution operation uses a transposition of this tensor, which is similar to the approach used in backpropagation (cf. Section 8.3.3). The counter-part deconvolution operation occurs from the sixth to the seventh layer (by counting the ReLU/pooling/unpooling layers in the middle). Therefore, we will define the (deconvolution) tensor $U^{(s,6)} = [u_{ijk}^{(s,6)}]$ in relation to $W^{(p,1)}$. Layer 5 contains d_2 feature maps, which were inherited from the convolution operation in the first layer (and unchanged by pooling/unpooling/ReLU operations). These d_2 feature maps need to be mapped into d_1 layers, where the value of d_1 is 3 for RGB color channels. Therefore, the number of filters in the deconvolution layer is equal to the depth of the filters in the convolution layer and vice vera. One can view this change in shape as a result of the transposition and spatial inversion of the 4-dimensional tensor created by the filters. Furthermore, the entries of the two 4-dimensional tensors are related as follows:

$$u_{ijk}^{(s,6)} = w_{rms}^{(k,1)} \quad \forall s \in \{1 \ldots d_1\}, \forall k \in \{1 \ldots d_2\} \tag{8.5}$$

Here, $r = n - i + 1$ and $m = n - j + 1$, where the spatial footprint in the first layer is $n \times n$. Note the transposition of the indices s and k in the above relationship. This relationship is identical to Equation 8.3. It is not necessary to tie the weights in the encoder and decoder, or even use a symmetric architecture between encoder and decoder [310].

The filters $U^{(s,6)}$ in the sixth layer are used just like any other convolution to reconstruct the RGB color channels of the images from the activations in layer 6. Therefore, a deconvolution operation is really a convolution operation, except that it is done with a transposed and spatially inverted filter. As discussed in Section 8.3.2, this type of deconvolution operation is also used in backpropagation. Both the convolution/deconvolution operations can also be executed with the use of matrix multiplications, as described in that section.

The pooling operations, however, irreversibly lose some information and are therefore impossible to invert exactly. This is because the non-maximal values in the layer are permanently lost by pooling. The max-unpooling operation is implemented with the help of *switches*. When pooling is performed, the precise positions of the maximal values are stored. For example, consider the common situation in which 2 × 2 pooling is performed at stride 2. In such a case, pooling reduces both spatial dimensions by a factor of 2, and it picks the maximum out of $2 \times 2 = 4$ values in each (non-overlapping) pooled region. The exact coordinate of the (maximal) value is stored, and is referred to as the switch. When unpooling, the dimensions are increased by a factor of 2, and the values at the switch positions are copied from the previous layer. The other values are set to 0. Therefore, after max-unpooling, exactly 75% of the entries in the layer will have uncopied values of 0 in the case of non-overlapping 2 × 2 pooling.

Like traditional autoencoders, the loss function is defined by the reconstruction error over all $L_1 \times L_1 \times d_1$ pixels. Therefore, if $h_{ijk}^{(1)}$ represents the values of the pixels in the first (input) layer, and $h_{ijk}^{(7)}$ represents the values of the pixels in the seventh (output) layer, the reconstruction loss E is defined as follows:

$$E = \sum_{i=1}^{L_1} \sum_{j=1}^{L_1} \sum_{k=1}^{d_1} (h_{ijk}^{(1)} - h_{ijk}^{(7)})^2 \tag{8.6}$$

Other types of error functions (such as L_1-loss and negative log-likelihood) are also used.

One can use traditional backpropagation with the autoencoder. Backpropagating through deconvolutions or the ReLU is no different than in the case of convolutions. In the case of max-unpooling, the gradient flows only through the switches in an unchanged way. Since the parameters of the encoder and the decoder are tied, one needs to sum up the gradients of the matching parameters in the two layers during gradient descent. Another interesting point is that backpropagating through deconvolutions uses almost identical operations to forward propagation through convolutions. This is because both backpropagation and deconvolution cause successive transpositions of the 4-dimensional tensor used for transformation.

This basic autoencoder can easily be extended to the case where multiple convolutions, poolings, and ReLUs are used. The work in [554] discusses the difficulty with multilayer autoencoders, and proposes several tricks to improve performance. There are several other architectural design choices that are often used to improve performance. One key point is that strided convolutions are often used (in lieu of max-pooling) to reduce the spatial footprint in the encoder, which must be balanced in the decoder with *fractionally* strided convolutions. Consider a situation in which the encoder uses a stride of S with some padding to reduce the size of the spatial footprint. In the decoder, one can increase the size of the spatial footprint by the same factor by using the following trick. While performing the convolution, we stretch the input volume by placing $S-1$ rows of zeros[9] between every pair of rows, and $S-1$ columns of zeros between every pair of columns before applying the filter. As a result, the input volume already stretches by a factor of approximately S in each spatial dimension. Additional padding along the borders can be applied before performing the convolution with the transposed filter. Such an approach has the effect of providing a fractional stride and expanding the output size in the decoder. An alternative approach for stretching the input volume of a convolution is to insert interpolated values (instead of zeros) between the original entries of the input volume. The interpolation is done using a convex combination of the nearest four values, and a decreasing function of the distance to each of these values is used as the proportionality factor of the interpolation [449]. The approach of stretching the inputs is sometimes combined with that of stretching the filters as well by inserting zeros within the filter [449]. Stretching the filter results in an approach, referred to as *dilated convolution*, although its use is not universal for fractionally strided convolutions. A detailed discussion of convolution arithmetic (included fractionally strided convolution) is provided in [109]. Compared to the traditional autoencoder, the convolutional autoencoder is somewhat more tricky to implement, with many different variations for better performance. Refer to the bibliographic nodes.

Unsupervised methods also have applications to improving supervised learning. The most obvious method among them is *pretraining*, which was discussed in Section 4.7 of Chapter 4. In convolutional neural networks, the methodology for pretraining is not very different in principle from what is used in traditional neural networks. Pretraining can also

[9] Example available at http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html.

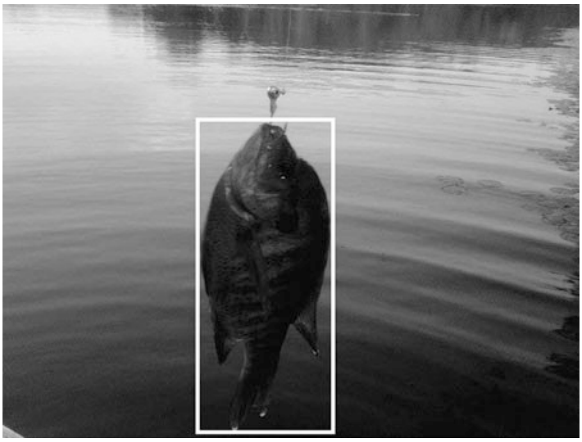


Figure 8.18: Example of image classification/localization in which the class "*fish*" is identified together with its bounding box. The image is illustrative only.

be performed by deriving the weights from a trained *deep-belief convolutional network* [285]. This is analogous to the approach in traditional neural networks, where stacked Boltzmann machines were among the earliest models used for pretraining.

8.6 Applications of Convolutional Networks

Convolutional neural networks have several applications in object detection, localization, video, and text processing. Many of these applications work on the basic principle of using convolutional neural networks to provide engineered features, on top of which multidimensional applications can be constructed. The success of convolutional neural networks remains unmatched by almost any class of neural networks. In recent years, competitive methods have even been proposed for sequence-to-sequence learning, which has traditionally been the domain of recurrent networks.

8.6.1 Content-Based Image Retrieval

In content-based image retrieval, each image is first engineered into a set of multidimensional features by using a pretrained classifier like *AlexNet*. The pretraining is typically done up front using a large data set like *ImageNet*. A huge number of choices of such pretrained classifiers is available at [586]. The features from the fully connected layers of the classifier can be used to create a multidimensional representation of the images. The multidimensional representations of the images can be used in conjunction with any multidimensional retrieval system to provide results of high quality. The use of neural codes for image retrieval is discussed in [16]. The reason that this approach works is because the features extracted from *AlexNet* have semantic significance to the different types of shapes present in the data. As a result, the quality of the retrieval is generally quite high when working with these features.

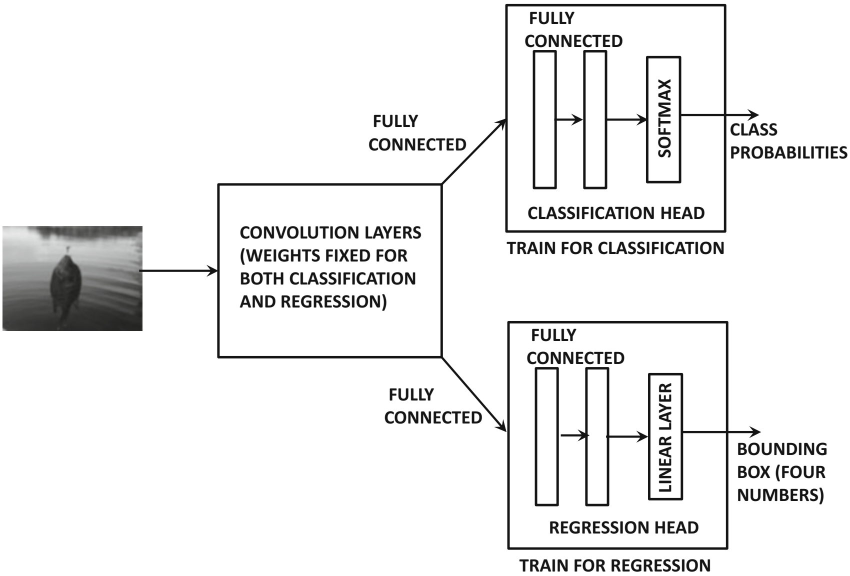


Figure 8.19: The broad framework of classification and localization

8.6.2 Object Localization

In object localization, we have a fixed set of objects in an image, and we would like to identify the rectangular regions in the image in which the object occurs. The basic idea is to take an image with a *fixed* number of objects and encase each of them in a bounding box. In the following, we will consider the simple case in which a single object exists in the image. Image localization is usually integrated with the classification problem, in which we first wish to classify the object in the image and draw a bounding box around it. For simplicity, we consider the case in which there is a single object in the image. We have shown an example of image classification and localization in Figure 8.18, in which the class "*fish*" is identified, and a bounding box is drawn around the portion of the image that delineates that class.

The bounding box of an image can be uniquely identified with four numbers. A common choice is to identify the top-left corner of the bounding box, and the two dimensions of the box. Therefore, one can identify a box with four unique numbers. This is a regression problem with multiple targets. Here, the key is to understand that one can train almost the same model for both classification and regression, which vary only in terms of the final two fully connected layers. This is because the semantic nature of the features extracted from the convolution network are often highly generalizable across a wide variety of tasks. Therefore, one can use the following approach:

1. First, we train a neural network classifier like *AlexNet* or use a pretrained version of this classifier. In the first phase, it suffices to train the classifier only with image-class pairs. One can even use an off-the-shelf pretrained version of the classifier, which was trained on *ImageNet*.

2. The last two fully connected layers and softmax layers are removed. This removed set of layers is referred to as the *classification head*. A new set of two fully connected

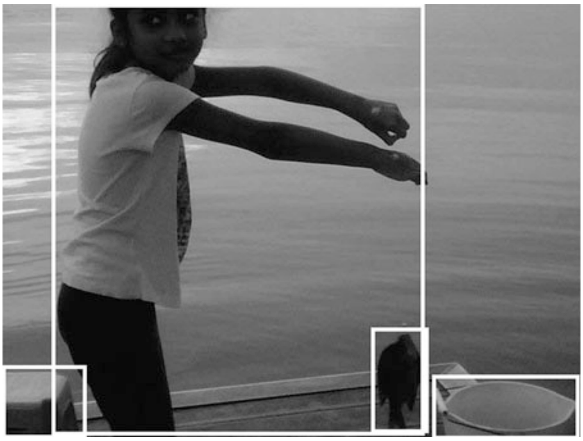


Figure 8.20: Example of object detection. Here, four objects are identified together with their bounding boxes. The four objects are "*fish*," "*girl*," "*bucket*," and "*seat*." The image is illustrative only.

layers and a linear regression layer is attached. Only these layers are then trained with training data containing images and their bounding boxes. This new set of layers is referred to as the *regression head*. Note that the weights of the convolution layers are fixed, and are not changed. Both the classification and regression heads are shown in Figure 8.19. Since the classification and regression heads are not connected to one another in any way, these two layers can be trained independently. The convolution layers play the role of creating visual features for both classification and regression.

3. One can optionally fine-tune the convolution layers to be sensitive to both classification and regression (since they were originally trained only for classification). In such a case, both classification and regression heads are attached, and the training data for images, their classes, and bounding boxes are shown to the network. Backpropagation is used to fine-tune all layers. This full architecture is shown in Figure 8.19.

4. The entire network (with both classification and regression heads attached) is then used on the test images. The outputs of the classification head provide the class probabilities, whereas the outputs of the regression head provide the bounding boxes.

One can obtain results of superior quality by using a sliding-window approach. The basic idea in the sliding-window approach is to perform the localization at multiple locations in the image with the use of a sliding window, and then integrate the results of the different runs. An example of this approach is the *Overfeat* method [441]. Refer to the bibliographic notes for pointers to other localization methods.

8.6.3 Object Detection

Object detection is very similar to object localization, except that there is a *variable* number of objects of different classes in the image. In this case, one wishes to identify all the objects in the image together with their classes. We have shown an example of object detection in Figure 8.20, in which there are four objects corresponding to the classes "*fish*," "*girl*," "*bucket*," and "*seat*." The bounding boxes of these classes are also shown in the figure.

Object detection is generally a more difficult problem than that of localization because of the variable number of outputs. In fact, one does not even know a priori how many objects there are in the image. For example, one cannot use the architecture of the previous section, where it is not clear how many classification or regression heads one might attach to the convolutional layers.

The simplest approach to this problem is to use a sliding window approach. In the sliding window approach, one tries all possible bounding boxes in the image, on which the object localization approach is applied to detect a single object. As a result, one might detect different objects in different bounding boxes, or the same object in overlapping bounding boxes. The detections from the different bounding boxes can then be integrated in order to provide the final result. Unfortunately, the approach can be rather expensive. For an image of size $L \times L$, the number of possible bounding boxes is L^4. Note that one would have to perform the classification/regression for each of these L^4 possibilities for each image at test time. This is a problem, because one generally expects the testing times to be modest enough to provide real-time responses.

In order to address this issue *region proposal methods* were advanced. The basic idea of a region proposal method is that it can serve as a general-purpose object detector that merges regions with similar pixels together to create larger regions. Therefore, the region proposal methods are used to first create a set of candidate bounding boxes, and then the object classification/localization method is run in each of them. Note that some candidate regions might not have valid objects, and others might have overlapping objects. These are then used to integrate and identify all the objects in the image. This broader approach has been used in various techniques like *MCG* [172], *EdgeBoxes* [568], and *SelectiveSearch* [501].

8.6.4 Natural Language and Sequence Learning

While the preferred way of machine learning with text sequences is that of recurrent neural networks, the use of convolutional neural networks has become increasingly popular in recent years. At first sight, convolutional neural networks do not seem like a natural fit for text-mining tasks. First, image shapes are interpreted in the same way, irrespective of where they are in the image. This is not quite the case for text, where the position of a word in a sentence seems to matter quite a bit. Second, issues such as position translation and shift cannot be treated in the same way in text data. Neighboring pixels in an image are usually very similar, whereas neighboring words in text are almost never the same. In spite of these differences, the systems based on convolutional networks have shown improved performance in recent years.

Just as an image is represented as a 2-dimensional object with an additional depth dimension defined by the number of color channels, a text sequence is represented as 1-dimensional object with depth defined by its dimensionality of representation. The dimensionality of representation of a text sentence is equal to the lexicon size for the case of one-hot encoding. Therefore, instead of 3-dimensional boxes with a spatial extent and a depth (color channels/feature maps), the filters for text data are 2-dimensional boxes with a window (sequence) length for sliding along the sentence and a depth defined by the lexicon. In later layers of the convolutional network, the depth is defined by the number of feature maps rather than the lexicon size. Furthermore, the number of filters in a given layer defines the number of feature maps in the next layer (as in image data). In image data, one performs convolutions at all 2-dimensional locations, whereas in text data one performs convolutions at all 1-dimensional points in the sentence with the same filter. One challenge with this approach is that the use of one-hot encoding increases the number of channels, and therefore blows up the number of parameters in the filters in the first layer. The lexicon size of a typical corpus may often be of the order of 10^6. Therefore, various types of pretrained embeddings of words, such as *word2vec* or *GLoVe* [371] are used (cf. Chapter 2) in lieu of the one-hot encodings of the individual words. Such word encodings are semantically rich, and the dimensionality of the representation can be reduced to a few thousand (from a hundred-thousand). This approach can provide an order of magnitude reduction in the number of parameters in the first layer, in addition to providing a semantically rich representation. All other operations (like max-pooling or convolutions) in the case of text data are similar to those of image data.

8.6.5 Video Classification

Videos can be considered generalizations of image data in which a temporal component is inherent to a sequence of images. This type of data can be considered *spatio-temporal data*, which requires us to generalize the 2-dimensional spatial convolutions to 3-dimensional spatio-temporal convolutions. Each frame in a video can be considered an image, and one therefore receives a sequence of images in time. Consider a situation in which each image is of size $224 \times 224 \times 3$, and a total of 10 frames are received. Therefore, the size of the video segment is $224 \times 224 \times 10 \times 3$. Instead of performing spatial convolutions with a 2-dimensional spatial filter (with an additional depth dimension capturing 3 color channels), we perform spatiotemporal convolutions with a 3-dimensional spatiotemporal filter (and a depth dimension capturing the color channels). Here, it is interesting to note that the nature of the filter depends on the data set at hand. A purely sequential data set (e.g., text) requires 1-dimensional convolutions with windows, an image data set requires 2-dimensional convolutions, and a video data set requires 3-dimensional convolutions. We refer to the bibliographic notes for pointers to several papers that use 3-dimensional convolutions for video classification.

An interesting observation is that 3-dimensional convolutions add only a limited amount to what one can achieve by averaging the classifications of individual frames by image classifiers. A part of the problem is that motion adds only a limited amount to the information that is available in the individual frames for classification purposes. Furthermore, sufficiently large video data sets are hard to come by. For example, even a data set containing a million videos is often not sufficient because the amount of data required for 3-dimensional convolutions is much larger than that required for 2-dimensional convolutions. Finally, 3-dimensional convolutional neural networks are good for relatively short segments of video (e.g., half a second), but they might not be so good for longer videos.

For the case of longer videos, it makes sense to combine recurrent neural networks (or LSTMs) with convolutional neural networks. For example, we can use 2-dimensional convolutions over individual frames, but a recurrent network is used to carry over states from one frame to the next. One can also use 3-dimensional convolutional neural networks over short segments of video, and then hook them up with recurrent units. Such an approach helps in identifying actions over longer time horizons. Refer to the bibliographic notes for pointers to methods that combine convolutional and recurrent neural networks.

8.7 Summary

This chapter discusses the use of convolutional neural networks with a primary focus on image processing. These networks are biologically inspired and are among the earliest success stories of the power of neural networks. An important focus of this chapter is the classification problem, although these methods can be used for additional applications such as unsupervised feature learning, object detection, and localization. Convolutional neural networks typically learn hierarchical features in different layers, where the earlier layers learn primitive shapes, whereas the later layers learn more complex shapes. The backpropagation methods for convolutional neural networks are closely related to the problems of deconvolution and visualization. Recently, convolutional neural networks have also been used for text processing, where they have shown competitive performance with recurrent neural networks.

8.8 Bibliographic Notes

The earliest inspiration for convolutional neural networks came from Hubel and Wiesel's experiments with the cat's visual cortex [212]. Based on many of these principles, the notion of the neocognitron was proposed in early work. These ideas were then generalized to the first convolutional network, which was referred to as *LeNet-5* [279]. An early discussion on the best practices and principles of convolutional neural networks may be found in [452]. An excellent overview of convolutional neural networks may be found in [236]. A tutorial on convolution arithmetic is available in [109]. A brief discussion of applications may be found in [283].

The earliest data set that was used popularly for training convolutional neural networks was the MNIST database of handwritten digits [281]. Later, larger datasets like *ImageNet* [581] became more popular. Competitions such as the *ImageNet* challenge (*ILSVRC*) [582] have served as sources of some of the best algorithms over the last five years. Examples of neural networks that have done well at various competitions include *AlexNet* [255], *ZFNet* [556], *VGG* [454], *GoogLeNet* [485], and *ResNet* [184]. The *ResNet* is closely related to highway networks [505], and it provides an iterative view of feature engineering. A useful precursor to *GoogLeNet* was the Network-in-Network (NiN) architecture [297], which illustrated some useful design principles of the inception module (such as the use of bottleneck operations). Several explanations of why *ResNet* works well are provided in [185, 505]. The use of inception modules between skip connections is proposed in [537]. The use of stochastic depth in combination with residual networks is discussed in [210]. Wide residual networks are proposed in [549]. A related architecture, referred to as *FractalNet* [268], uses both short and long paths in the network, but does not use skip connections. Training is done by dropping subpaths in the network, although prediction is done on the full network.

Off-the-shelf feature extraction methods with pretrained models are discussed in [223, 390, 585]. In cases where the nature of the application is very different from *ImageNet* data, it might make sense to extract features only from the lower layers of the pretrained model. This is because lower layers often encode more generic/primitive features like edges and basic shapes, which tend to work across an array of settings. The local-response normalization approach is closely related to the contrast normalization discussed in [221].

The work in [466] proposes that it makes sense to replace the max-pooling layer with a convolutional layer with increased stride. Not using a max-pooling layer is an advantage in the construction of an autoencoder because one can use a convolutional layer with a fractional stride within the decoder [384]. Fractional strides place zeros within the rows and columns of the input volume, when it is desired to increase the spatial footprint from the convolution operation. The notion of *dilated convolutions* [544] in which zeros are placed within the rows/columns of the filter (instead of input volume) is also sometimes used. The connections between deconvolution networks and gradient-based visualization are discussed in [456, 466]. Simple methods for inverting the features created by a convolutional neural network are discussed in [104]. The work in [308] discuss how to reconstruct an image optimally from a given feature representation. The earliest use the convolutional autoencoder is discussed in [387]. Several variants of the basic autoencoder architecture were proposed in [318, 554, 555]. One can also borrow ideas from restricted Boltzmann machines to perform unsupervised feature learning. One of the earliest such ideas that uses Deep Belief Nets (DBNs) is discussed in [285]. The use of different types of deconvolution, visualization, and reconstruction is discussed in [130, 554, 555, 556]. A very large-scale study for unsupervised feature extraction from images is reported in [270].

There are some ways of learning feature representations in an unsupervised way, which seem to work quite well. The work in [76] clusters on small image patches with a k-means algorithm in order to generate features. The centroids of the clusters can be used to extract features. Another option is use random weights as filters in order to extract features [85, 221, 425]. Some insight on this issue is provided in [425], which shows that a combination of convolution and pooling becomes frequency selective and translation invariant, even with random weights.

A discussion of neural feature engineering for image retrieval is provided in [16]. Numerous methods have been proposed in recent years for image localization. A particularly prominent system in this regard was *Overfeat* [441], which was the winner of the 2013 *ImageNet* competition. This method used a sliding-window approach in order to obtain results of superior quality. Variations of *AlexNet*, *VGG*, and *ResNet* have also done well in the *ImageNet* competition. Some of the earliest methods for object detection were proposed in [87, 117]. The latter is also referred to as the *deformable parts model* [117]. These methods did not use neural networks or deep learning, although some connections have been drawn [163] between deformable parts models and convolutional neural networks. In the deep learning era, numerous methods like *MCG* [172], *EdgeBoxes* [568], and *SelectiveSearch* [501] have been proposed. The main problem with these methods is that they are somewhat slow. Recently, the *Yolo* method, which is a fast object detection method, was proposed in [391]. However, some of the speed gains are at the expense of accuracy. Nevertheless, the overall effectiveness of the method is still quite high. The use of convolutional neural networks for image segmentation is discussed in [180]. Texture synthesis and style transfer methods with convolutional neural networks are proposed in [131, 132, 226]. Tremendous advances have been made in recent years in facial recognition with neural networks. The early work [269, 407] showed how convolutional networks can be used for face recognition. Deep variants are discussed in [367, 474, 475].

Convolutional neural networks for natural language processing are discussed in [78, 79, 102, 227, 240, 517]. These methods often leverage on *word2vec* or GloVe methods to start with a richer set of features [325, 371]. The notion of recurrent and convolutional neural networks has also been combined for text classification [260]. The use of character-level convolutional networks for text classification is discussed in [561]. Methods for image captioning by combining convolutional and recurrent neural networks are discussed in [225, 509]. The use of convolutional neural networks for processing graph-structured data is discussed in [92, 188, 243]. A discussion of the use of convolutional neural networks in time-series and speech is provided [276].

Video data can be considered the spatiotemporal generalization of image data from the perspective of convolutional networks [488]. The use of 3-dimensional convolutional neural networks for large-scale video classification is discussed in [17, 222, 234, 500], and the works in [17, 222] proposed the earliest methods for 3-dimensional convolutional neural networks in video classification. All the neural networks for image classification have natural 3-dimensional counterparts. For example, a generalization of *VGG* to the video domain with 3-dimensional convolutional networks is discussed in [500]. Surprisingly, the results from 3-dimensional convolutional networks are only slightly better than single-frame methods, which perform classifications from individual frames of the video. An important observation is that individual frames already contain a lot of information for classification purposes, and the addition of motion often does not help for classification, unless the motion characteristics are essential for distinguishing classes. Another issue is that the data sets for video classification are often limited in scale compared to what is really required for building large-scale systems. Even though the work in [234] collected a relatively large-scale data set of over a million *YouTube* videos, this scale seems to be insufficient in the context of video processing. After all, video processing requires 3-dimensional convolutions that are far more complex than the 2-dimensional convolutions in image processing. As a result, it is often beneficial to combine hand-crafted features with the convolutional neural network [514]. Another useful feature that has found applicability in recent years is the notion of optical flow [53]. The use of 3-dimensional convolutional neural networks is helpful for classification of videos over shorter time scales. Another common idea for video classification is to combine convolutional neural networks with recurrent neural networks [17, 100, 356, 455]. The work in [17] was the earliest method for combining recurrent and convolutional neural networks. The use of recurrent neural networks is helpful when one has to perform the classification over longer time scales. A recent method [21] combines recurrent and convolutional neural networks in a homogeneous way. The basic idea is to make every neuron in the convolution neural network to be recurrent. One can view this approach to be a direct recurrent extension of convolutional neural networks.

8.8.1 Software Resources and Data Sets

A variety of packages are available for deep learning with convolutional neural networks like *Caffe* [571], *Torch* [572], *Theano* [573], and *TensorFlow* [574]. Extensions of *Caffe* to Python and MATLAB are available. A discussion of feature extraction from *Caffe* may be found in [585]. A "model zoo" of pretrained models from *Caffe* may be found in [586]. *Theano* is Python-based, and it provides high-level packages like *Keras* [575] and *Lasagne* [576] as interfaces. An open-source implementation of convolutional neural networks in MATLAB, referred to as *MatConvNet*, may be found in [503]. The code and parameter files for *AlexNet* are available at [584].

The two most popular data sets for testing convolutional neural networks are *MNIST* and *ImageNet*. Both these data sets are described in detail in Chapter 1. The *MNIST* data set is quite well behaved because its images have been centered and normalized. As a result, the images in *MNIST* can be classified accurately even with conventional machine learning methods, and therefore convolutional neural networks are not necessary. On the other hand, the images in *ImageNet* contain images from different perspectives, and do require convolutional neural networks. Nevertheless, the 1000-category setting of *ImageNet*, together with its large size, makes it a difficult candidate for testing in a computationally efficient way. A more modestly sized data set is *CIFAR-10* [583]. This data set contains only 60,000 instances divided into ten categories, and contains 6,000 color images. Each image in the data set has size $32 \times 32 \times 3$. It is noteworthy that the *CIFAR-10* data set is a small subset of the *tiny images data set* [642], which originally contains 80 million images. The *CIFAR-10* data set is often used for smaller scale testing, before a more large-scale training is done with *ImageNet*. The *CIFAR-100* data set is just like the *CIFAR-10* data set, except that it has 100 classes, and each class contains 600 instances. The 100 classes are grouped into 10 super-classes.

8.9 Exercises

1. Consider a 1-dimensional time-series with values 2, 1, 3, 4, 7. Perform a convolution with a 1-dimensional filter 1, 0, 1 and zero padding.

2. For a one-dimensional time series of length L and a filter of size F, what is the length of the output? How much padding would you need to keep the output size to a constant value?

3. Consider an activation volume of size $13 \times 13 \times 64$ and a filter of size $3 \times 3 \times 64$. Discuss whether it is possible to perform convolutions with strides 2, 3, 4, and 5. Justify your answer in each case.

4. Work out the sizes of the spatial convolution layers for each of the columns of Table 8.2. In each case, we start with an input image volume of $224 \times 224 \times 3$.

5. Work out the number of parameters in each spatial layer for column D of Table 8.2.

6. Download an implementation of the *AlexNet* architecture from a neural network library of your choice. Train the network on subsets of varying size from the *ImageNet* data, and plot the top-5 error with data size.

7. Compute the convolution of the input volume in the upper-left corner of Figure 8.2 with the horizontal edge detection filter of Figure 8.1(b). Use a stride of 1 without padding.

8. Perform a 4×4 pooling at stride 1 of the input volume in the upper-left corner of Figure 8.4.

9. Discuss the various type of pretraining that one can use in the image captioning application discussed in Section 7.7.1 of Chapter 7.

10. You have a lot of data containing ratings of users for different images. Show how you can combine a convolutional neural network with the collaborative filtering ideas discussed in Chapter 2 to create a hybrid between a collaborative and content-centric recommender system.