Digital Image Processing, 4th Ed.

Neural Networks and Deep Learning
Deep Convolutional Neural Networks

this parameter to $\alpha = 0.1$ resulted in a drop of the best correct recognition rate to 49.1%. Based on the preceding results, we used $\alpha = 0.001$ and 50,000 epochs to train the network.

The parameters in Fig. 12.38 were the result of training. The recognition rate for the training data using these parameters was 97%. We achieved a recognition rate of 95.6% on the test set using the same parameters. The difference between these two figures, and the 96.4% and 96.2%, respectively, obtained for the same data with the Bayes classifier (see Example 12.6), are statistically insignificant.

The fact that our neural networks achieved results comparable to those obtained with the Bayes classifier is not surprising. It can be shown (Duda, Hart, and Stork [2001]) that a three-layer neural net, trained by backpropagation using a sum of errors squared criterion, approximates the Bayes decision functions in the limit, as the number of training samples approaches infinity. Although our training sets were small, the data were well behaved enough to yield results that are close to what theory predicts.

12.6 DEEP CONVOLUTIONAL NEURAL NETWORKS

Up to this point, we have organized pattern features as vectors. Generally, this assumes that the form of those features has been specified (i.e., "engineered" by a human designer) and extracted from images prior to being input to a neural network (Example 12.13 is an illustration of this approach). But one of the strengths of neural networks is that they are capable of learning pattern features directly from training data. What we would like to do is input a set of training images directly into a neural network, and have the network learn the necessary features *on its own*. One way to do this would be to convert images to vectors directly by organizing the pixels based on a linear index (see Fig. 12.1), and then letting each element (pixel) of the linear index be an element of the vector. However, this approach does not utilize any spatial relationships that may exist between pixels in an image, such as pixel arrangements into corners, the presence of edge segments, and other features that may help to differentiate one image from another. In this section, we present a class of neural networks called *deep convolutional neural networks* (*CNNs* or *ConvNets* for short) that accept images as inputs and are ideally suited for automatic learning and image classification. In order to differentiate between CNNs and the neural nets we studied in Section 12.5, we will refer to the latter as "fully connected" neural networks.

A BASIC CNN ARCHITECTURE

In the following discussion, we use a *LeNet* architecture (see references at the end of this chapter) to introduce convolutional nets. We do this for two main reasons: First, the LeNet architecture is reasonably simple to understand. This makes it ideal for introducing basic CNN concepts. Second, our real interest is in deriving the equations of backpropagation for convolutional networks, a task that is simplified by the intuitiveness of LeNets.

To simplify the explanation of the CNN in Fig. 12.40, we focus attention initially on a single image input. Multiple input images are a trivial extension we will consider later in our discussion.

The CNN in Fig. 12.40 contains all the basic elements of a LeNet architecture, and we use it without loss of generality. A key difference between this architecture and the neural net architectures we studied in the previous section is that inputs to CNNs are 2-D arrays (images), while inputs to our fully connected neural networks are vectors. However, as you will see shortly, the computations performed by both networks are very similar: (1) a sum of products is formed, (2) a bias value is added,

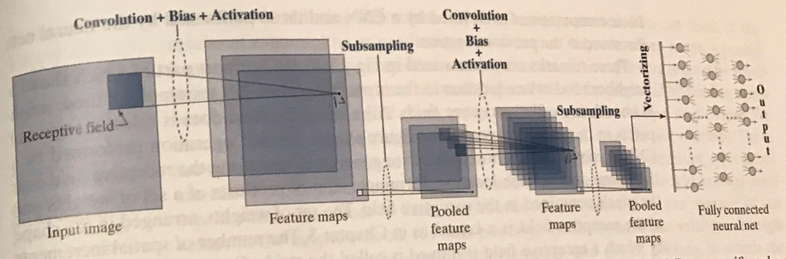


FIGURE 12.40 A CNN containing all the basic elements of a LeNet architecture. Points A and B are specific values to be addressed later in this section. The last pooled feature maps are vectorized and serve as the input to a fully connected neural network. The class to which the input image belongs is determined by the output neuron with the highest value.

(3) the result is passed through an activation function, and (4) the activation value becomes a single input to a following layer.

Despite the fact that the computations performed by CNNs and fully connected neural nets are similar, there are some basic differences between the two, beyond their input formats being 2-D versus vectors. An important difference is that CNNs are capable of learning 2-D features directly from raw image data, as mentioned earlier. Because the tools for systematically engineering comprehensive feature sets for complex image recognition tasks do not exist, having a system that can learn its own image features from raw image data is a crucial advantage of CNNs. Another major difference is in the way in which layers are connected. In a fully connected neural net, we feed the output of *every* neuron in a layer directly into the input of *every* neuron in the next layer. By contrast, in a CNN we feed into every input of a layer, a *single* value, determined by the convolution (hence the name *convolutional* neural net) over a spatial neighborhood in the output of the previous layer. Therefore, CNNs are not fully connected in the sense defined in the last section. Another difference is that the 2-D arrays from one layer to the next are subsampled to reduce sensitivity to translational variations in the input. These differences and their meaning will become clear as we look at various CNN configurations in the following discussion.

We will discuss in the next subsection the exact form of neural computations in a CNN, and show they are equivalent in form to the computations performed by neurons in a fully connected neural net.

Basics of How a CNN Operates

As noted above, the type of neighborhood processing in CNNs is spatial convolution. We explained the mechanics of spatial convolution in Fig. 3.29, and expressed it mathematically in Eq. (3-35). As that equation shows, convolution computes a sum of products between pixels and a set of kernel weights. This operation is carried out at every spatial location in the input image. The result at each location (x, y) in the input is a scalar value. Think of this value as the output of a neuron in a layer of a fully connected neural net. If we add a bias and pass the result through an activation function (see Fig. 12.29), we have a complete analogy between the

basic computations performed by a CNN and those performed by the neural nets discussed in the previous section.

These remarks are summarized in Fig. 12.40, the leftmost part of which shows a neighborhood at one location in the input image. In CNN terminology, these neighborhoods are called *receptive fields*. All a receptive field does is select a region of pixels in the input image. As the figure shows, the first operation performed by a CNN is convolution, whose values are generated by moving the receptive field over the image and, at each location, forming a sum of products of a set of weights and the pixels contained in the receptive field. The set of weights, arranged in the shape of the receptive field, is a *kernel*, as in Chapter 3. The number of spatial increments by which a receptive field is moved is called the *stride*. Our spatial convolutions in previous chapters had a stride of one, but that is not a requirement of the equations themselves. In CNNs, an important motivations for using strides greater than one is data reduction. For example, changing the stride from one to two reduces the image resolution by one-half in each spatial dimension, resulting in a three-fourths reduction in the amount of data per image. Another important motivation is as a substitute for subsampling which, as we discuss below, is used to reduce system sensitivity to spatial translation.

To each convolution value (sum of products) we add a bias, then pass the result through an activation function to generate a single value. Then, this value is fed to the corresponding (x, y) location in the input of the next layer. When repeated for all locations in the input image, the process just explained results in a 2-D set of values that we store in next layer as a 2-D array, called a *feature map*. This terminology is motivated by the fact that the role performed by convolution is to extract features such as edges, points, and blobs from the input (remember, convolution is the basis of spatial filtering, which we used in Chapter 3 for tasks such as smoothing, sharpening, and computing edges in an image). The same weights and a single bias are used to generate the convolution (feature map) values corresponding to all locations of the receptive field in the input image. This is done to cause the same feature to be detected at all points in the image. Using the same weights and bias for this purpose is called *weight* (or *parameter*) *sharing*.

In the terminology of Chapter 3, a feature map is a spatially filtered image.

Figure 12.40 shows three feature maps in the first layer of the network. The other two feature maps are generated in the manner just explained, but using a *different* set of weights and bias for each feature map. Because each set of weights and bias is different, each feature map generally will contain a different set of features, all extracted from the same input image. The feature maps are referred to collectively as a *convolutional layer*. Thus, the CNN in Fig. 12.40 has two convolutional layers.

The process after convolution and activation is *subsampling* (also called *pooling*), which is motivated by a model of the mammal visual cortex proposed by Hubel and Wiesel [1959]. Their findings suggest that parts of the visual cortex consist of *simple* and *complex* cells. The simple cells perform feature extraction, while the complex cells combine (aggregate) those features into a more meaningful whole. In this model, a reduction in spatial resolution appears to be responsible for achieving translational invariance. Pooling is a way of modeling this reduction in dimensionality. When training a CNN with large image databases, pooling has the additional

advantage of reducing the volume of data being processed. You can think of the results of subsampling as producing *pooled feature maps*. In other words, a pooled feature map is a feature map of reduced spatial resolution. Pooling is done by subdividing a feature map into a set of small (typically 2×2) regions, called *pooling neighborhoods*, and replacing *all* elements in such a neighborhood by a *single* value. We assume that pooling neighborhoods are *adjacent* (i.e., they do not overlap). There are several ways to compute the pooled values; collectively, the different approaches are called *pooling methods*. Three common pooling methods are: (1) *average pooling*, in which the values in each neighborhood are replaced by the average of the values in the neighborhood; (2) *max-pooling*, which replaces the values in a neighborhood by the maximum value of its elements; and (3) L_2 pooling, in which the resulting pooled value is the square root of the sum of the neighborhood values squared. There is one pooled feature map for each feature map. The pooled feature maps are referred to collectively as a *pooling layer*. In Fig. 12.40 we used 2×2 pooling so each resulting pooled map is one-fourth the size of the preceding feature map. The use of receptive fields, convolution, parameter sharing, and pooling are characteristics unique to CNNs.

Adjacency is not a requirement of pooling per se. We assume it here for simplicity and because this is an approach that is used frequently.

Because feature maps are the result of spatial convolution, we know from Chapter 3 that they are simply filtered images. It then follows that pooled feature maps are filtered images of lower resolution. As Fig. 12.40 illustrates, the pooled feature maps in the first layer become the inputs to the next layer in the network. But, whereas we showed a single image as an input to the first layer, we now have multiple pooled feature maps (filtered images) that are inputs into the second layer.

To see how these multiple inputs to the second layer are handled, focus for a moment on one pooled feature map. To generate the values for the first feature map in the second convolutional layer, we perform convolution, add a bias, and use activation, as before. Then, we change the kernel and bias, and repeat the procedure for the second feature map, still using the same input. We do this for every remaining feature map, changing the kernel weights and bias for each. Then, we consider the next pooled feature map input and perform the same procedure (convolution, plus bias, plus activation) for every feature map in the second layer, using yet another set of different kernels and biases. When we are finished, we will have generated three values for the same location in every feature map, with one value coming from the corresponding location in each of the three inputs. The question now is: How do we combine these three individual values into one? The answer lies in the fact that convolution is a linear process, from which it follows that the three individual values are combined into one by superposition (that is, by adding them).

You could interpret the convolution with several input images as 3-D convolution, but with movement only in the spatial (x and y) directions. The result would be identical to summing individual convolutions with each image separately, as we do here.

In the first layer, we had one input image and three feature maps, so we needed three kernels to complete all required convolutions. In the second layer, we have three inputs and seven feature maps, so the total number of kernels (and biases) needed is $3 \times 7 = 21$. Each feature map is pooled to generate a corresponding pooled feature map, resulting in seven pooled feature maps. In Fig. 12.40, there are only two layers, so these seven pooled feature maps are the outputs of the last layer.

As usual, the ultimate objective is to use features for classification, so we need a classifier. As Fig. 12.40 shows, in a CNN we perform classification by feeding the

value of the last pooled layer into a fully connected neural net, the details of which you learned in Section 12.5. But the outputs of a CNN are 2-D arrays (i.e., filtered images of reduced resolution), whereas the inputs to a fully connected net are vectors. Therefore, we have to *vectorize* the 2-D pooled feature maps in the last layer. We do this using linear indexing (see Fig. 12.1). Each 2-D array in the last layer of the CNN is converted into a vector, then all resulting vectors are concatenated (vertically for a column) to form a single vector. This vector propagates through the neural net, as explained in Section 12.5. In any given application, the number of outputs in the fully connected net is equal to the number of pattern classes being classified. As before, the output with the highest value determines the class of the input.

The parameters of the fully connected neural net are learned during training of the CNN, to be discussed shortly.

EXAMPLE 12.14: Receptive fields, pooling neighborhoods, and their corresponding feature maps.

The top row of Fig. 12.41 shows a numerical example of the relative sizes of feature maps and pooled feature maps as a function of the sizes of receptive fields and pooling neighborhoods. The input image is of size 28×28 pixels, and the receptive field is of size 5×5. If we require that the receptive field be contained in the image during convolution, you know from Section 3.4 that the resulting convolution array (feature map) will be of size 24×24. If we use a pooling neighborhood of size 2×2, the resulting pooled feature maps will be of size 12×12, as the figure shows. As noted earlier, we assume that pooling neighborhoods do not overlap.

As an analogy with fully connected neural nets, think of each element of a 2-D array in the top row of Fig. 12.41 as a neuron. The outputs of the neurons in the input are pixel values. The neurons in the feature map of the first layer have output values generated by convolving with the input image a kernel whose size and shape are the same as the receptive field, and whose coefficients are learned during training. To each convolution value we add a bias and pass the result through an activation function to generate the output value of the corresponding neuron in the feature map. The output values of the neurons in the pooled feature maps are generated by pooling the output values of the neurons in the feature maps.

The second row in Fig. 12.41 illustrates visually how feature maps and pooled feature maps look based on the input image shown in the figure. The kernel shown is as described in the previous paragraph, and its weights (shown as intensity values) were learned from sample images using the training of the CNN described later in Example 12.17. Therefore, the nature of the learned features is determined by the learned kernel coefficients. Note that the contents of the feature maps are specific features detected by convolution. For example, some of the features emphasize edges in the the character. As mentioned earlier, the pooled features are lower-resolution versions of this effect.

EXAMPLE 12.15: Graphical illustration of the functions performed by the components of a CNN.

Figure 12.42 shows the 28×28 image from Fig. 12.41, input into an expanded version of the CNN architecture from Fig. 12.40. The expanded CNN, which we will discuss in more detail in Example 12.17, has six feature maps in the first layer, and twelve in the second. It uses receptive fields of size 5×5, and pooling neighborhoods of size 2×2. Because the receptive fields are of size 5×5, the feature maps in the first layer are of size 24×24, as we explained in Example 12.14. Each feature map has its own set of weights and bias, so we will need a total of $(5 \times 5) \times 6 + 6 = 156$ parameters (six kernels with twenty-five weights each, and six biases) to generate the feature maps in the first layer. The top row of Fig. 12.43(a) shows the kernels with the weights learned during training of the CNN displayed as images, with intensity being proportional to kernel values.

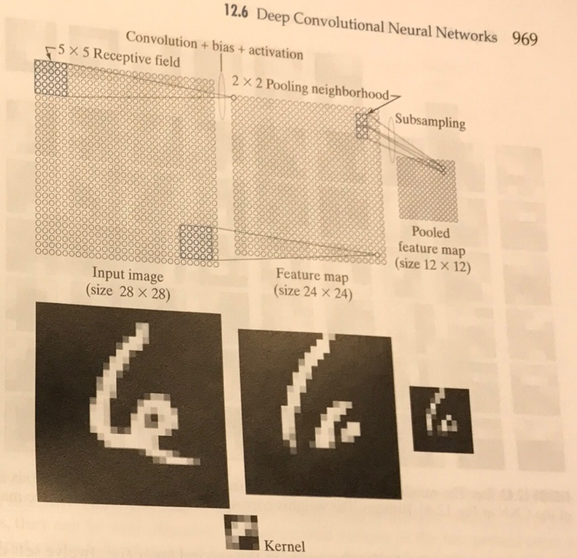


FIGURE 12.41 Top row: How the sizes of receptive fields and pooling neighborhoods affect the sizes of feature maps and pooled feature maps. Bottom row: An image example. This figure is explained in more detail in Example 12.17. (Image courtesy of NIST.)

Because we used pooling neighborhoods of size 2×2, the pooled feature maps in the first layer of Fig. 12.42 are of size 12×12. As we discussed earlier, the number of feature maps and pooled feature maps is the same, so we will have six arrays of size 12×12 acting as inputs to the twelve feature maps in the second layer (the number of feature maps generally is different from layer to layer). Each feature map will have its own set of weights and bias, so will need a total of $6 \times (5 \times 5) \times 12 + 12 = 1812$

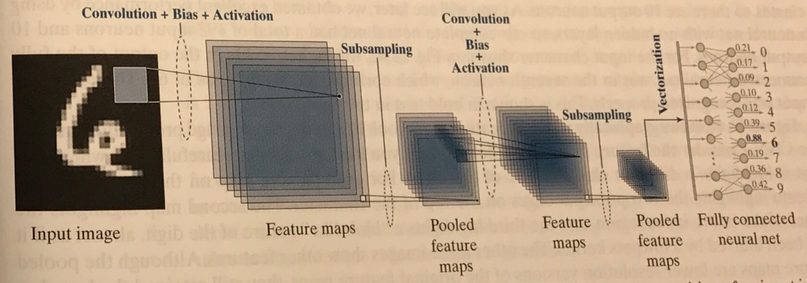


FIGURE 12.42 Numerical example illustrating the various functions of a CNN, including recognition of an input image. A sigmoid activation function was used throughout.

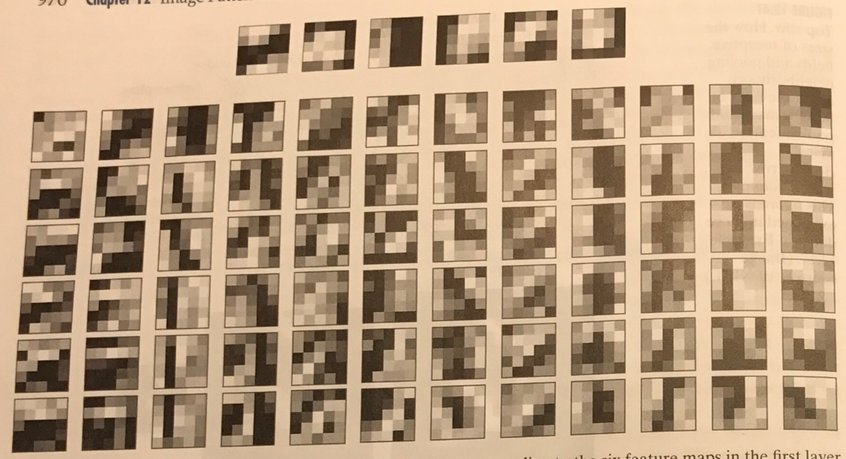


FIGURE 12.43 Top: The weights (shown as images of size 5×5) corresponding to the six feature maps in the first layer of the CNN in Fig. 12.42. Bottom: The weights corresponding to the twelve feature maps in the second layer.

parameters to generate the feature maps in the second layer (i.e., twelve sets of six kernels with twenty-five weights each, plus twelve biases). The bottom part of Fig. 12.43 shows the kernels as images. Because we are using receptive fields of size 5×5, the feature maps in the second layer are of size 8×8. Using 2×2 pooling neighborhoods resulted in pooled feature maps of size 4×4 in the second layer.

As we discussed earlier, the pooled feature maps in the last layer have to be vectorized to be able to input them into the fully connected neural net. Each pooled feature map resulted in a column vector of size 16×1. There are 12 of these vectors which, when concatenated vertically, resulted in a single vector of size 192×1. Therefore, our fully connected neural net has 192 input neurons. There are ten numeral classes, so there are 10 output neurons. As you will see later, we obtained excellent performance by using a neural net with no hidden layers, so our complete neural net had a total of 192 input neurons and 10 output neurons. For the input character shown in Fig. 12.42, the highest value in the output of the fully connected neural net was in the seventh neuron, which corresponds to the class of 6's. Therefore, the input was recognized properly. This is shown in bold text in the figure.

Figure 12.44 shows graphically what the feature maps look like as the input image propagates through the CNN. Consider the feature maps in the first layer. If you look at each map carefully, you will notice that it highlights a different characteristic of the input. For example, the map on the top of the first column highlights the two principal edges on the top of the character. The second map highlights the edges of the entire inner region, and the third highlights a "blob-like" nature of the digit, almost as if it had been blurred by a lowpass kernel. The other three images show other features. Although the pooled feature maps are lower-resolution versions of the original feature maps, they still retained the key characteristics of the features in the latter. If you look at the first two feature maps in the second layer, and compare them with the first two in the first layer, you can see that they could be interpreted as higher-

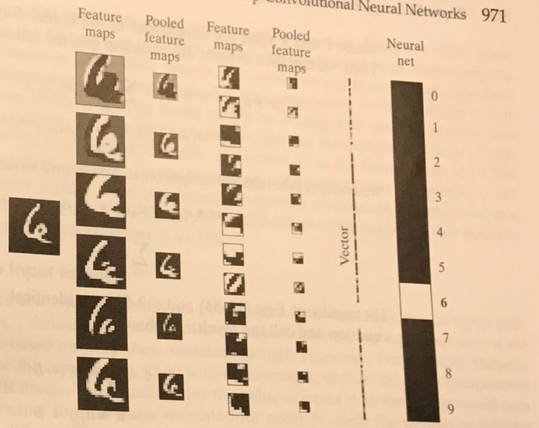


FIGURE 12.44 Visual summary of an input image propagating through the CNN in Fig. 12.42. Shown as images are all the results of convolution (feature maps) and pooling (pooled feature maps) for both layers of the network. (Example 12.17 contains more details about this figure.)

level abstractions of the top part of the character, in the sense that they show an area flanked on both sides by areas of opposite intensity. These abstractions are not always easy to analyze visually, but as you will see in later examples, they can be very effective. The vectorized version of the last pooled layer is self-explanatory. The output of the fully connected neural net shows dark for low values and white for the highest value, indicating that the input was properly recognized as a number 6. Later in this section, we will show that the simple CNN architecture in Fig. 12.42 is capable of recognizing the correct class of over 70,000 numerical samples with nearly perfect accuracy.

Neural Computations in a CNN

Recall from Fig. 12.29 that the basic computation performed by an artificial neuron is a sum of products between weights and values from a previous layer. To this we add a bias and call the result the *net* (*total*) *input* to the neuron, which we denoted by z_i. As we showed in Eq. (12-54), the sum involved in generating z_i is a single sum. The computations performed in a CNN to generate a single value in a feature map is 2-D convolution. As you learned in Chapter 3, this is a double sum of products between the coefficients of a kernel and the corresponding elements of the image array overlapped by the kernel. With reference to Fig. 12.40, let w denote a kernel formed by arranging the weights in the shape of the receptive field we discussed in connection with that figure. For notational consistency with Section 12.5, let $a_{x,y}$ denote image or pooled feature values, depending on the layer. The convolution value at any point (x, y) in the input is given by

$$w \star a_{x,y} = \sum_l \sum_k w_{l,k} a_{x-l,y-k} \tag{12-83}$$

where l and k span the dimensions of the kernel. Suppose that w is of size 3×3. Then, we can then expand this equation into the following sum of products:

$$\begin{aligned} w \star a_{x,y} &= w \star a_{x,y} = \sum_l \sum_k w_{l,k} a_{x-l,y-k} \\ &= w_{1,1} a_{x-1,y-1} + w_{1,2} a_{x-1,y-2} + \cdots + w_{3,3} a_{x-3,y-3} \end{aligned} \tag{12-84}$$

We could relabel the subscripts on w and a, and write instead

$$\begin{aligned} w \star a_{x,y} &= w_1 a_1 + w_2 a_2 + \cdots + w_9 a_9 \\ &= \sum_{i=1}^{9} w_i a_i \end{aligned} \tag{12-85}$$

The results of Eqs. (12-84) and (12-85) are identical. If we add a bias to the latter equation and call the result z we have

$$\begin{aligned} z &= \sum_{j=1}^{9} w_j a_j + b \\ &= w \star a_{x,y} + b \end{aligned} \tag{12-86}$$

The form of the first line of this equation is identical to Eq. (12-54). Therefore, we conclude that if we add a bias to the spatial convolution computation performed by a CNN at any fixed position (x, y) in the input, the result can be expressed in a form identical to the computation performed by an artificial neuron in a fully connected neural net. We need the x, y only to account for the fact that we are working in 2-D. If we think of z as the net input to a neuron, the analogy with the neurons discussed in Section 12.5 is completed by passing z through an activation function, h, to get the output of the neuron:

$$a = h(z) \tag{12-87}$$

This is exactly how the value of any point in a feature map (such as the point labeled A in Fig. 12.40) is computed.

Now consider point B in that figure. As mentioned earlier, its value is given by adding three convolution equations:

$$\begin{aligned} w_{l,k}^{(1)} \star a_{x,y}^{(1)} + w_{l,k}^{(2)} \star a_{x,y}^{(2)} + w_{l,k}^{(3)} \star a_{x,y}^{(3)} = &\sum_l \sum_k w_{l,k}^{(1)} a_{x-l,y-k}^{(1)} + \\ &\sum_l \sum_k w_{l,k}^{(2)} a_{x-l,y-k}^{(2)} + \sum_l \sum_k w_{l,k}^{(2)} a_{x-l,y-k}^{(2)} \end{aligned} \tag{12-88}$$

where the superscripts refer to the three pooled feature maps in Fig. 12.40. The values of l, k, x, and y are the same in all three equations because all three kernels are of the same size and they move in unison. We could expand this equation and obtain a sum of products that is lengthier than for point A in Fig. 12.40, but we could still relabel all terms and obtain a sum of products that involves only one summation, *exactly* as before.

The preceding result tells us that the equations used to obtain the value of an element of any feature map in a CNN can be expressed in the form of the computation performed by an artificial neuron. This holds for any feature map, regardless of how many convolutions are involved in the computation of the elements of that feature map, in which case we would simply be dealing with the sum of more convolution equations. The implication is that we can use the basic form of Eqs. (12-86) and (12-87) to describe how the value of an element in any feature map of a CNN is obtained. This means we do not have to account explicitly for the number of different pooled feature maps (and hence the number of different kernels) used in a pooling layer. The result is a significant simplification of the equations that describe forward and backpropagation in a CNN.

Multiple Input Images

The values of $a_{x,y}$ just discussed are pixel values in the first layer but, in layers past the first, $a_{x,y}$ denotes values of pooled features. However, our equations do not differentiate based on what these variables actually represent. For example, suppose we replace the input to Fig. 12.40 with three images, such as the three components of an RGB image. The equations for the value of point A in the figure would now have the same form as those we stated for point B—only the weights and biases would be different. Thus, the results in the previous discussion for one input image are applicable directly to multiple input images. We will give an example of a CNN with three input images later in our discussion.

THE EQUATIONS OF A FORWARD PASS THROUGH A CNN

We concluded in the preceding discussion that we can express the result of convolving a kernel, w, and an input array with values $a_{x,y}$, as

$$\begin{aligned} z_{x,y} &= \sum_l \sum_k w_{l,k} a_{x-l,y-k} + b \\ &= w \star a_{x,y} + b \end{aligned} \tag{12-89}$$

As noted earlier, a kernel is formed by organizing the weights in the shape of a corresponding receptive field. Also keep in mind that w and $a_{x,y}$ represent *all* the weights and corresponding values in a set of input images or pooled features.

where l and k span the dimensions of the kernel, x and y span the dimensions of the input, and b is a bias. The corresponding value of $a_{x,y}$ is

$$a_{x,y} = h\left(z_{x,y}\right) \tag{12-90}$$

But this $a_{x,y}$ is different from the one we used to compute Eq. (12-89), in which $a_{x,y}$ represents values from the previous layer. Thus, we are going to need additional notation to differentiate between layers. As in fully connected neural nets, we use ℓ for this purpose, and write Eqs. (12-89) and (12-90) as

$$\begin{aligned} z_{x,y}(\ell) &= \sum_l \sum_k w_{l,k}(\ell) a_{x-l,y-k}(\ell-1) + b(\ell) \\ &= w(\ell) \star a_{x,y}(\ell-1) + b(\ell) \end{aligned} \tag{12-91}$$

and

$$a_{x,y}(\ell) = h\left(z_{x,y}(\ell)\right) \tag{12-92}$$

for $\ell = 1, 2, \ldots, L_c$, where L_c is the number of convolutional layers, and $a_{x,y}(\ell)$ denotes the values of pooled features in convolutional layer ℓ. When $\ell = 1$,

$$a_{x,y}(0) = \{\text{values of pixels in the input image(s)}\} \tag{12-93}$$

When $\ell = L_c$,

$$a_{x,y}(L_c) = \{\text{values of pooled features in last layer of the CNN}\} \tag{12-94}$$

Note that ℓ starts at 1 instead of 2, as we did in Section 12.5. The reason is that we are naming layers, as in "convolutional layer ℓ." It would be confusing to start at convolutional layer 2. Finally, we note that the pooling does not require any convolutions. The only function of pooling is to reduce the spatial dimensions of the feature map preceding it, so we do not include explicit pooling equations here.

Equations (12-91) through (12-94) are all we need to compute all values in a forward pass through the convolutional section of a CNN. As described in Fig. 12.40, the values of the pooled features of the last layer are vectorized and fed into a fully connected feedforward neural network, whose forward propagation is explained in Eqs. (12-54) and (12-55) or, in matrix form, in Table 12.2.

THE EQUATIONS OF BACKPROPAGATION USED TO TRAIN CNNs

As you saw in the previous section, the feedforward equations of a CNN are similar to those of a fully connected neural net, but with multiplication replaced by convolution, and notation that reflects the fact that CNNs are not fully connected in the sense defined in Section 12.5. As you will see in this section, the equations of backpropagation also are similar in many respects to those in fully connected neural nets.

As in the derivation of backpropagation in Section 12.5, we start with the definition of how the output error of our CNN changes with respect to each neuron in the network. The form of the error is the same as for fully connected neural nets, but now it is a function of x and y instead of j:

$$\delta_{x,y}(\ell) = \frac{\partial E}{\partial z_{x,y}(\ell)} \tag{12-95}$$

As in Section 12.5, we want to relate this quantity to $\delta_{xy}(\ell+1)$, which we again do using the chain rule:

$$\delta_{x,y}(\ell) = \frac{\partial E}{\partial z_{x,y}(\ell)} = \sum_u \sum_v \frac{\partial E}{\partial z_{u,v}(\ell+1)} \frac{\partial z_{u,v}(\ell+1)}{\partial z_{x,y}(\ell)} \tag{12-96}$$

where u and v are any two variables of summation over the range of possible values of z. As noted in Section 12.5, these summations result from applying the chain rule.

By definition, the first term of the double summation of Eq. (12-96) is $\delta_{x,y}(\ell+1)$. So, we can write this equation as

$$\delta_{x,y}(\ell) = \frac{\partial E}{\partial z_{x,y}(\ell)} = \sum_u \sum_v \delta_{u,v}(\ell+1) \frac{\partial z_{u,v}(\ell+1)}{\partial z_{x,y}(\ell)} \tag{12-97}$$

Substituting Eq. (12-92) into Eq. (12-91), and using the resulting $z_{u,v}$ in Eq. (12-97), we obtain

$$\delta_{x,y}(\ell) = \sum_u \sum_v \delta_{u,v}(\ell+1) \frac{\partial}{\partial z_{x,y}(\ell)} \left[\sum_l \sum_k w_{l,k}(\ell+1) h\left(z_{u-l,v-k}(\ell)\right) + b(\ell+1) \right] \tag{12-98}$$

The derivative of the expression inside the brackets is zero unless $u - l = x$ and $v - k = y$, and because the derivative of $b(\ell+1)$ with respect to $z_{x,y}(\ell)$ is zero. But, if $u - l = x$ and $v - k = y$, then $l = u - x$ and $k = v - y$. Therefore, taking the indicated derivative of the expression in brackets, we can write Eq. (12-98) as

$$\delta_{x,y}(\ell) = \sum_u \sum_v \delta_{u,v}(\ell+1) \left[\sum_{u-x} \sum_{v-y} w_{u-x,v-y}(\ell+1) h'\left(z_{x,y}(\ell)\right) \right] \tag{12-99}$$

Values of x, y, u, and v are specified outside of the terms inside the brackets. Once the values of these variables are fixed, $u - x$ and $v - y$ inside the brackets are simply two *constants*. Therefore, the double summation evaluates to $w_{u-x,v-y}(\ell+1) h'\left(z_{x,y}(\ell)\right)$, and we can write Eq. (12-99) as

$$\begin{aligned} \delta_{x,y}(\ell) &= \sum_u \sum_v \delta_{u,v}(\ell+1) w_{u-x,v-y}(\ell+1) h'\left(z_{x,y}(\ell)\right) \\ &= h'\left(z_{x,y}(\ell)\right) \sum_u \sum_v \delta_{u,v}(\ell+1) w_{u-x,v-y}(\ell+1) \end{aligned} \tag{12-100}$$

The double sum expression in the second line of this equation is in the form of a convolution, but the displacements are the negatives of those in Eq. (12-91). Therefore, we can write Eq. (12-100) as

$$\delta_{x,y}(\ell) = h'\left(z_{x,y}(\ell)\right) \left[\delta_{x,y}(\ell+1) \star w_{-x,-y}(\ell+1) \right] \tag{12-101}$$

The negatives in the subscripts indicate that w is *reflected* about both spatial axes. This is the same as rotating w by 180°, as we explained in connection with Eq. (3-35). Using this fact, we finally arrive at an expression for the error at a layer ℓ by writing Eq. (12-101) equivalently as

The 180° rotation is for *each* 2-D kernel in a layer.

$$\delta_{x,y}(\ell) = h'\left(z_{x,y}(\ell)\right) \left[\delta_{x,y}(\ell+1) \star \text{rot180}\left(w_{x,y}(\ell+1)\right) \right] \tag{12-102}$$

But the kernels do not depend on x and y, so we can write this equation as

$$\delta_{x,y}(\ell) = h'\left(z_{x,y}(\ell)\right)\left[\delta_{x,y}(\ell+1) \star \text{rot180}\left(w(\ell+1)\right)\right] \tag{12-103}$$

As in Section 12.5, our final objective is to compute the change in E with respect to the weights and biases. Following a similar procedure as above, we obtain

$$\begin{aligned}
\frac{\partial E}{\partial w_{l,k}} &= \sum_x \sum_y \frac{\partial E}{\partial z_{x,y}(\ell)} \frac{\partial z_{x,y}(\ell)}{\partial w_{l,k}} \\
&= \sum_x \sum_y \delta_{x,y}(\ell) \frac{\partial z_{x,y}(\ell)}{\partial w_{l,k}} \\
&= \sum_x \sum_y \delta_{x,y}(\ell) \frac{\partial}{\partial w_{l,k}} \left[\sum_l \sum_k w_{l,k}(\ell) h\left(z_{x-l,y-k}(\ell-1)\right) + b(\ell) \right] \\
&= \sum_x \sum_y \delta_{x,y}(\ell) h\left(z_{x-l,y-k}(\ell-1)\right) \\
&= \sum_x \sum_y \delta_{x,y}(\ell) a_{x-l,y-k}(\ell-1)
\end{aligned} \tag{12-104}$$

where the last line follows from Eq. (12-92). This line is in the form of a convolution but, comparing it to Eq. (12-91), we see there is a sign reversal between the summation variables and their corresponding subscripts. To put it in the form of a convolution, we write the last line of Eq. (12-104) as

$$\begin{aligned}
\frac{\partial E}{\partial w_{l,k}} &= \sum_x \sum_y \delta_{x,y}(\ell) a_{-(l-x),-(k-y)}(\ell-1) \\
&= \delta_{l,k}(\ell) \star a_{-l,-k}(\ell-1) \\
&= \delta_{l,k}(\ell) \star \text{rot180}\left(a(\ell-1)\right)
\end{aligned} \tag{12-105}$$

Similarly (see Problem 12.32),

$$\frac{\partial E}{\partial b(\ell)} = \sum_x \sum_y \delta_{x,y}(\ell) \tag{12-106}$$

Using the preceding two expressions in the gradient descent equations (see Section 12.5), it follows that

$$\begin{aligned}
w_{l,k}(\ell) &= w_{l,k}(\ell) - \alpha \frac{\partial E}{\partial w_{l,k}} \\
&= w_{l,k}(\ell) - \alpha \delta_{l,k}(\ell) \star \text{rot180}\left(a(\ell-1)\right)
\end{aligned} \tag{12-107}$$

and

$$\begin{aligned}
b(\ell) &= b(\ell) - \alpha \frac{\partial E}{\partial b(\ell)} \\
&= b(\ell) - \alpha \sum_x \sum_y \delta_{x,y}(\ell)
\end{aligned} \tag{12-108}$$

Equations (12-107) and (12-108) update the weights and bias of each convolution layer in a CNN. As we have mentioned before, it is understood that the $w_{l,k}$ represents *all* the weights of a layer. The variables l and k span the spatial dimensions of the 2-D kernels, all of which are of the same size.

In a forward pass, we went from a convolution layer to a pooled layer. In backpropagation, we are going in the opposite direction. But the pooled feature maps are smaller than their corresponding feature maps (see Fig. 12.40). Therefore, when going in the reverse direction, we *upsample* (e.g., by pixel replication) each pooled feature map to match the size of the feature map that generated it. Each pooled feature map corresponds to a unique feature map, so the path of backpropagation is clearly defined.

With reference to Fig. 12.40, backpropagation starts at the output of the fully connected neural net. We know from Section 12.5 how to update the weights of this network. When we get to the "interface" between the neural net and the CNN, we have to reverse the vectorization method used to generate input vectors. That is, before we can proceed with backpropagation using Eqs. (12-107) and (12-108), we have to *regenerate* the individual pooled feature maps from the single vector propagated back by the fully connected neural net.

We summarized in Table 12.3 the backpropagation steps for a fully connected neural net. Table 12.6 summarizes the steps for performing backpropagation in the CNN architecture in Fig. 12.40. The procedure is repeated for a specified number of

TABLE 12.6
The principal steps used to train a CNN. The network is initialized with a set of small random weights and biases. In backpropagation, a vector arriving (from the fully connected net) at the output pooling layer must be converted to 2-D arrays of the same size as the pooled feature maps in that layer. Each pooled feature map is upsampled to match the size of its corresponding feature map. The steps in the table are for one epoch of training.

Step	Description	Equations
Step 1	Input images	$a(0)$ = the set of image pixels in the input to layer 1
Step 2	Forward pass	For each neuron corresponding to location (x,y) in each feature map in layer ℓ compute: $z_{x,y}(\ell) = w(\ell) \star a_{x,y}(\ell-1) + b(\ell)$ and $a_{x,y}(\ell) = h\left(z_{x,y}(\ell)\right)$; $\ell = 1, 2, \ldots, L_c$
Step 3	Backpropagation	For each neuron in each feature map in layer ℓ compute: $\delta_{x,y}(\ell) = h'\left(z_{x,y}(\ell)\right)\left[\delta_{x,y}(\ell+1) \star \text{rot180}\left(w(\ell+1)\right)\right]$; $\ell = L_c - 1, L_c - 2, \ldots, 1$
Step 4	Update parameters	Update the weights and bias for each feature map using $w_{l,k}(\ell) = w_{l,k}(\ell) - \alpha \delta_{l,k}(\ell) \star \text{rot180}\left(a(\ell-1)\right)$ and $b(\ell) = b(\ell) - \alpha \sum_x \sum_y \delta_{x,y}(\ell)$; $\ell = 1, 2, \ldots, L_c$

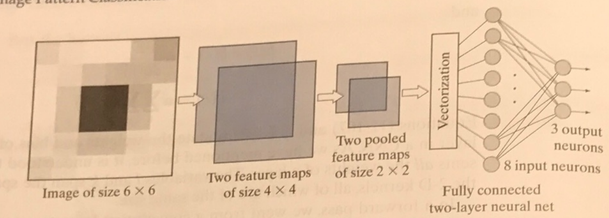


FIGURE 12.45 CNN with one convolutional layer used to learn to recognize the images in Fig. 12.46.

epochs, or until the output error of the neural net reaches an acceptable value. The error is computed exactly as we did in Section 12.5. It can be the mean squared error, or the recognition error. Keep in mind that the weights in $w(\ell)$ and the bias value $b(\ell)$ are *different* for *each* feature map in layer ℓ.

EXAMPLE 12.16: Teaching a CNN to recognize some simple images.

We begin our illustrations of CNN performance by teaching the CNN in Fig. 12.45 to recognize the small 6×6 images in Fig. 12.46. As you can see on the left of this figure, there are three samples each of images of a horizontal stripe, a small centered square, and a vertical stripe. These images were used as the training set. On the right are noisy samples of images in these three categories. These were used as the test set.

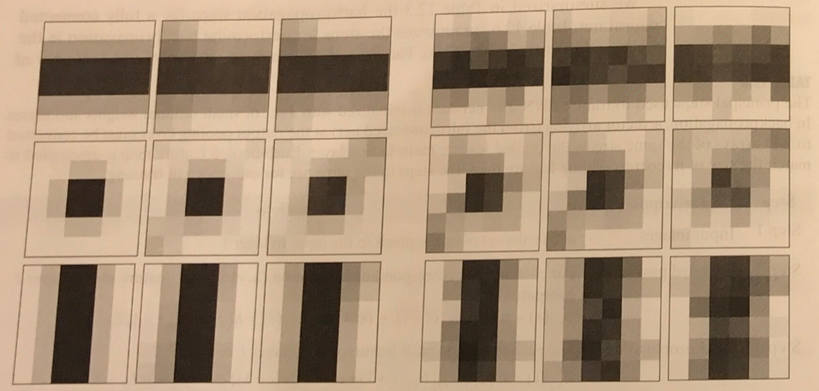


FIGURE 12.46 *Left:* Training images. Top row: Samples of a dark horizontal stripe. Center row: Samples of a centered dark square. Bottom row: Samples of a dark vertical stripe. Right: Noisy samples of the three categories on the left, created by adding Gaussian noise of zero mean and unit variance to the samples on the left. (All images are 8-bit grayscale images.)

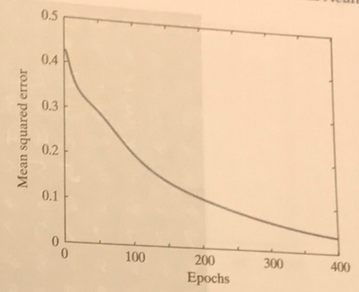


FIGURE 12.47 Training MSE as a function of epoch for the images in Fig. 12.46. Perfect recognition of the training and test sets was achieved after approximately 100 epochs, despite the fact that the MSE was relatively high there.

As Fig. 12.45 shows, the inputs to our system are single images. We used a receptor field of size 3×3, which resulted in feature maps of size 4×4. There are two feature maps, which means we need two kernels of size 3×3, and two biases. The pooled feature maps were generated using average pooling in neighborhoods of size 2×2. This resulted in two pooled feature maps of size 2×2, because the feature maps are of size 4×4. The two pooled maps contain eight total elements which were organized as an 8-D column vector to vectorize the output of the last layer. (We used linear indexing of each image, then concatenated the two resulting 4-D vectors into a single 8-D vector.) This vector was then fed into the fully connected neural net on the right, which consists of the input layer and a three-neuron output layer, one neuron per class. Because this network has no hidden layers, it implements linear decision functions (see Problem 12.18). To train the system, we used $\alpha = 1.0$ and ran the system for 400 epochs. Figure 12.47 is a plot of the MSE as a function of epoch. Perfect recognition of the training set was achieved after approximately 100 epochs of training, despite the fact that the MSE was relatively high there. Recognition of the test set was 100% as well. The kernel and bias values learned by the system were:

$$\boldsymbol{w}_1 = \begin{bmatrix} 3.0132 & 1.1808 & -0.0945 \\ 0.9718 & 0.7087 & -0.9093 \\ 0.7193 & 0.0230 & -0.8833 \end{bmatrix}, b_1 = -0.2990 \quad \boldsymbol{w}_2 = \begin{bmatrix} -0.7388 & 1.8832 & 4.1077 \\ -1.0027 & 0.3908 & 2.0357 \\ -1.2164 & -1.1853 & -0.1987 \end{bmatrix}, b_2 = -0.2834$$

It is important that the CNN learned these parameters automatically from the raw training images. No features in the sense discussed in Chapter 11 were employed.

EXAMPLE 12.17: Using a large training set to teach a CNN to recognize handwritten numerals.

In this example, we look at a more practical application using a database containing 60,000 training and 10,000 test images of handwritten numeric characters. The content of this database, called the *MNIST database*, is similar to a database from NIST (National Institute of Standards and Technology). The former is a "cleaned up" version of the latter, in which the characters have been centered and formatted into grayscale images of size 28×28 pixels. Both databases are freely available online. Figure 12.48 shows examples of typical numeric characters available in the databases. As you can see, there is

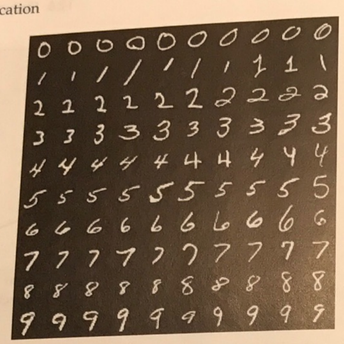


FIGURE 12.48 Samples similar to those available in the NIST and MNIST databases. Each character subimage is of size 28×28 pixels. (Individual images courtesy of NIST.)

significant variability in the characters—and this is just a small sampling of the 70,000 characters available for experimentation.

Figure 12.49 shows the architecture of the CNN we trained to recognize the ten digits in the MNIST database. We trained the system for 200 epochs using $\alpha = 1.0$. Figure 12.50 shows the training MSE as a function of epoch for the 60,000 training images in the MNIST database.

Training was done using mini batches of 50 images at a time to improve the learning rate (see the discussion in Section 12.7). We also classified all images of the training set and all images of the test set after each epoch of training. The objective of doing this was to see how quickly the system was learning the characteristics of the data. Figure 12.51 shows the results. A high level of correct recognition performance was achieved after relatively few epochs for both data sets, with approximately 98% correct recognition achieved after about 40 epochs. This is consistent with the training MSE in Fig. 12.50, which dropped quickly, then began a slow descent after about 40 epochs. Another 160 epochs of training were required for the system to achieve recognition of about 99.9%. These are impressive results for such a small CNN.

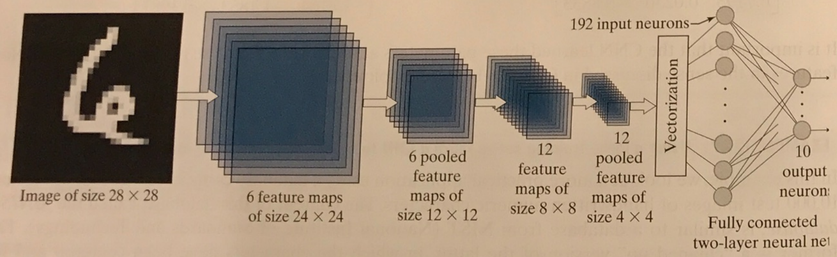


FIGURE 12.49 CNN used to recognize the ten digits in the MNIST database. The system was trained with 60,000 numerical character images of the same size as the image shown on the left. This architecture is the same as the architecture we used in Fig. 12.42. (Image courtesy of NIST.)

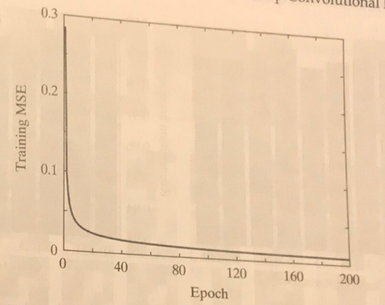
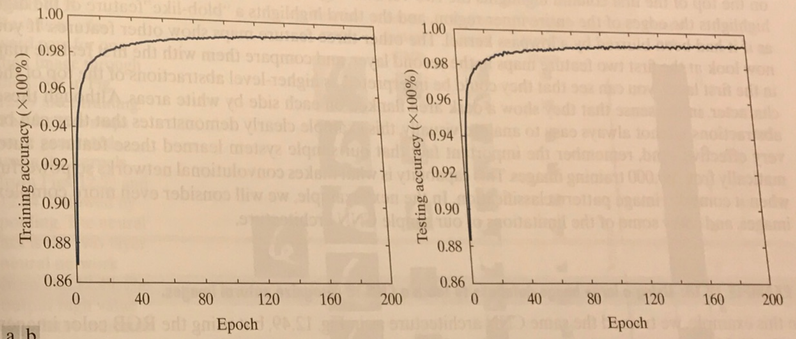


FIGURE 12.50 Training mean squared error as a function of epoch for the 60,000 training digit images in the MNIST database.

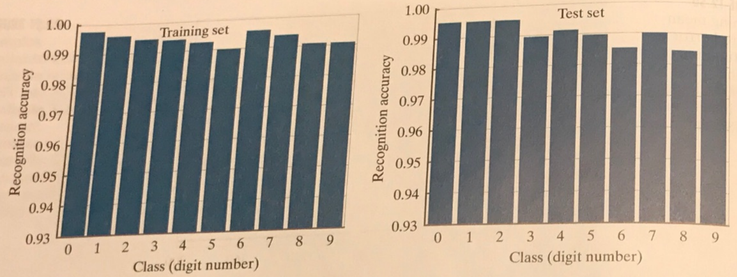
Figure 12.52 shows recognition performance on each digit class for both the training and test sets. The most revealing feature of these two graphs is that the CNN did equally as well on both sets of data. This is a good indication that the training was successful, and that it generalized well to digits it had not seen before. This is an example of the neural network not "over-fitting" the data in the training set.

Figure 12.53 shows the values of the kernels for the first feature map, displayed as intensities. There is one input image and six feature maps, so six kernels are required to generate the feature maps of the first layer. The dimensions of the kernels are the same as the receptive field, which we set at 5×5. Thus, the first image on the left in Fig. 12.53 is the 5×5 kernel corresponding to the first feature map. Figure 12.54 shows the kernels for the second layer. In this layer, we have six inputs (which are the pooled maps of the first layer) and twelve feature maps, so we need a total of $6 \times 12 = 72$ kernels and biases to generate the twelve feature maps in the second layer. Each column of Fig. 12.54 shows the six



a b

FIGURE 12.51 (a) Training accuracy (percent correct recognition of the training set) as a function of epoch for the 60,000 training images in the MNIST database. The maximum achieved was 99.36% correct recognition. (b) Accuracy as a function of epoch for the 10,000 test images in the MNIST database. The maximum correct recognition rate was 99.13%.



a b

FIGURE 12.52 (a) Recognition accuracy of training set by image class. Each bar shows a number between 0 and 1. When multiplied by 100%, these numbers give the correct recognition percentage for that class. (b) Recognition results per class in the test set. In both graphs the recognition rate is above 98%.

5×5 kernels corresponding to one of the feature maps in the second layer. We used 2×2 pooling in both layers, resulting in a 50% reduction of each of the two spatial dimensions of the feature maps.

Finally, it is of interest to visualize how one input image proceeds through the network, using the kernels learned during training. Figure 12.55 shows an input digit image from the test set, and the computations performed by the CNN at each layer. As before, we display numerical results as intensities.

Consider the results of convolution in the first layer. If you look at each resulting feature map carefully, you will notice that it highlights a different characteristic of the input. For example, the feature map on the top of the first column highlights the two vertical edges on the top of the character. The second highlights the edges of the entire inner region, and the third highlights a "blob-like"feature of the digit, as if it had been blurred by a lowpass kernel. The other three feature maps show other features. If you now look at the first two feature maps in the second layer, and compare them with the first feature map in the first layer, you can see that they could be interpreted as higher-level abstractions of the top of the character, in the sense that they show a dark area flanked on each side by white areas. Although these abstractions are not always easy to analyze visually, this example clearly demonstrates that they can be very effective. And, remember the important fact that our simple system learned these features automatically from 60,000 training images. This capability is what makes convolutional networks so powerful when it comes to image pattern classification. In the next example, we will consider even more complex images, and show some of the limitations of our simple CNN architecture.

EXAMPLE 12.18: Using a large image database to teach a CNN to recognize natural images.

In this example, we trained the same CNN architecture as in Fig. 12.49, but using the RGB color images in Fig. 12.56. These images are representative of those found in the CIFAR-10 database, a popular database used to test the performance of image classification systems. Our objective was to test the limitations of the CNN architecture in Fig. 12.49 by training it with data that is significantly more complex than the MNIST images in Example 12.17. The only difference between the architecture needed to

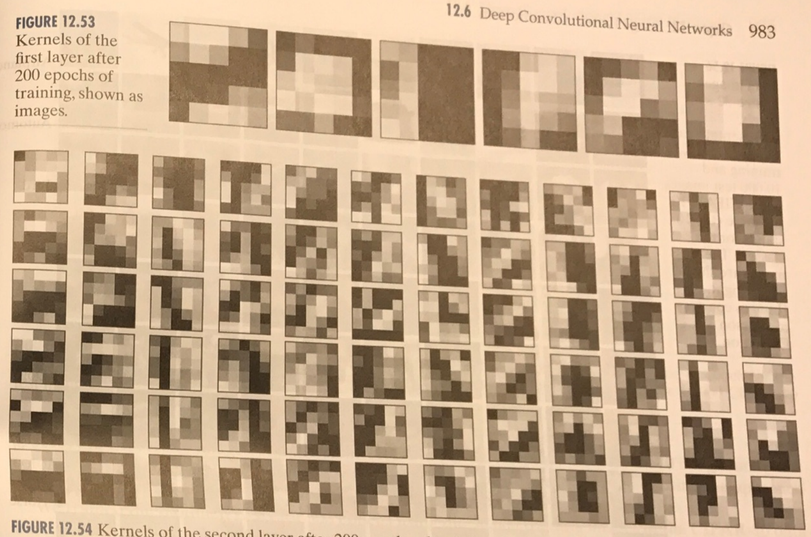


FIGURE 12.53 Kernels of the first layer after 200 epochs of training, shown as images.

FIGURE 12.54 Kernels of the second layer after 200 epochs of training, displayed as images of size 5×5. There are six inputs (pooled feature maps) into the second layer. Because there are twelve feature maps in the second layer, the CNN learned the weights of $6 \times 12 = 72$ kernels.

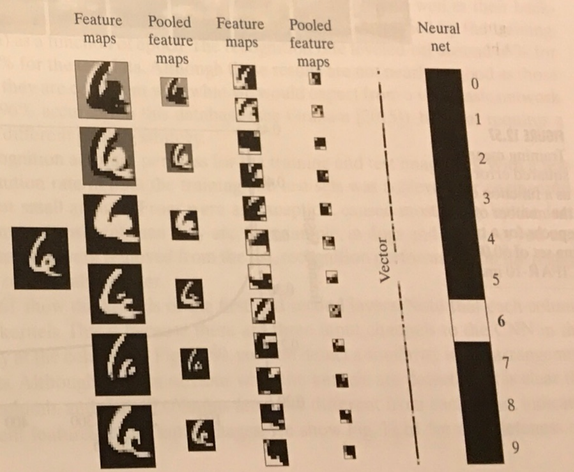


FIGURE 12.55 Results of a forward pass for one digit image through the CNN in Fig. 12.49 after training. The feature maps were generated using the kernels from Figs. 12.53 and 12.54, followed by pooling. The neural net is the two-layer neural network from Fig. 12.49. The output high value (in white) indicates that the CNN recognized the input properly. (This figure is the same as Fig. 12.44.)

FIGURE 12.56 Mini images of size 32×32 pixels, representative of the 50,000 training and 10,000 test images in the CIFAR-10 database (the 10 stands for ten classes). The class names are shown on the right. (Images courtesy of Pearson Education.)

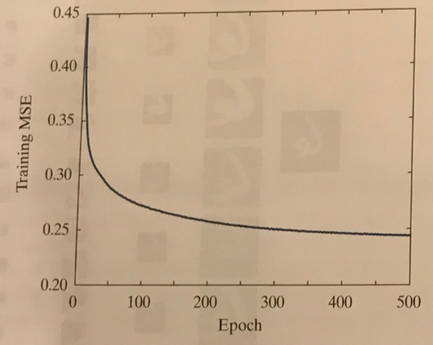
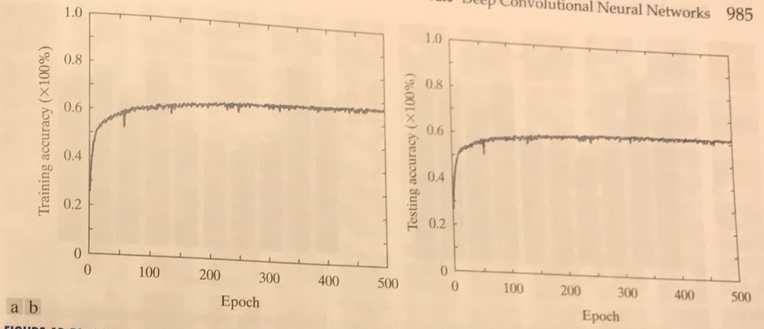


FIGURE 12.57 Training mean squared error as a function of the number of epochs for a training set of 50,000 CIFAR-10 images.



a b

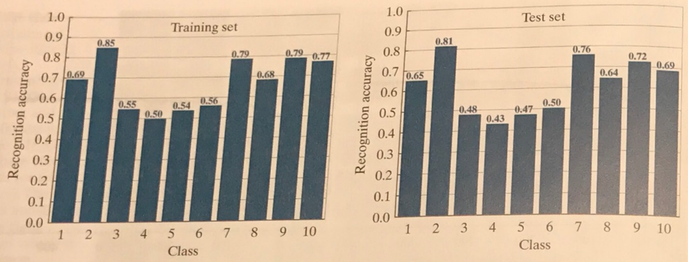
FIGURE 12.58 (a) Training accuracy (percent correct recognition of the training set) as a function of epoch for the 50,000 training images in the CIFAR-10 database. (b) Accuracy as a function of epoch for the 10,000 CIFAR-10 test images.

process the CIFAR-10 images, and the architecture in Fig. 12.49, is that the CIFAR-10 images are RGB color images, and hence have three channels. We worked with these input images using the approach explained in the subsection entitled Multiple Input Images, on page 973.

We trained the modified CNN for 500 epochs using the 50,000 training images of the CIFAR-10 database. Figure 12.57 is a plot of the mean squared error as a function of epoch during the training phase. Observe that the MSE begins to plateau at a value of approximately 0.25. In contrast, the MSE plot in Fig. 12.50 for the MNIST data achieved a much lower final value. This is not unexpected, given that the CIFAR-10 images are significantly more complex, both in the objects of interest as well as their backgrounds. The lower expected recognition performance of the training set is confirmed by the training-accuracy plotted in Fig. 12.58(a) as a function of epoch. The recognition rate leveled-off around 68% for the training data and about 61% for the test data. Although these results are not nearly as good as those obtained for the MNIST data, they are consistent with what we would expect from a very basic network. It is possible to achieve over 96% accuracy on this database (see Graham [2015]), but that requires a more complex network and a different pooling strategy.

Figure 12.59 shows the recognition accuracy per class for the training and test image sets. With a few exceptions, the highest recognition rate in both the training and test sets was achieved for engineered objects, and the lowest was for small animals. Frogs were an exception, caused most likely by the fact that frog size and shape are more consistent than they are, for example, in dogs and birds. As you can see in Fig. 12.59, if the small animals were removed from the list, recognition performance on the rest of the images would have been considerably higher.

Figures 12.60 and Fig. 12.61 show the kernels of the first and second layers. Note that each column in Fig. 12.60 has three 5×5 kernels. This is because there are three input channels to the CNN in this example. If you look carefully at the columns in Fig. 12.60, you can detect a similarity in the arrangement and values of the coefficients. Although it is not obvious what the kernels are detecting, it is clear that they are consistent in each column, and that all columns are quite different from each other, indicating a capability to detect different features in the input images. We show Fig. 12.61 for completeness only,

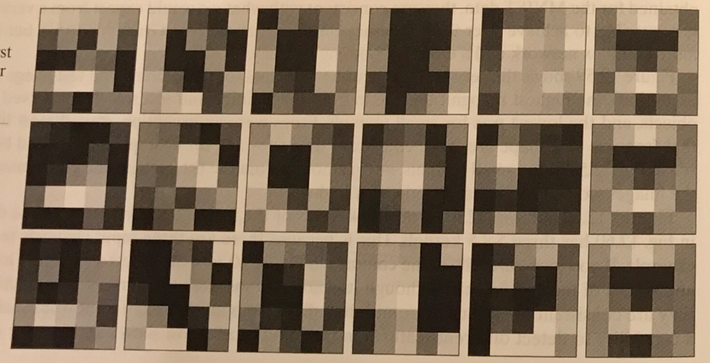


a b

FIGURE 12.59 (a) CIFAR-10 recognition rate of training set by image class. Each bar shows a number between 0 and 1. When multiplied by 100%, these numbers give the correct recognition percentage for that class. (b) Recognition results per class in the test set.

as there is little we can infer that deep into the network, especially at this small scale, and considering the complexity of the images in the training set. Finally, Fig. 12.62 shows a complete recognition pass through the CNN using the weights in Figs. 12.60 and 12.61. The input shows the three color channels of the RGB image in the seventh column of the first row in Fig. 12.56. The feature maps in the first column, show the various features extracted from the input. The second column shows the pooling results, zoomed to the size of the features maps for clarity. The third and fourth columns show the results in the second layer, and the fifth column shows the vectorized output. Finally, the last column shows the result of recognition, with white representing a high output, and the others showing much smaller values. The input image was properly recognized as belonging to class 1.

FIGURE 12.60 Weights of the kernels of the first convolution layer after 500 epochs of training.



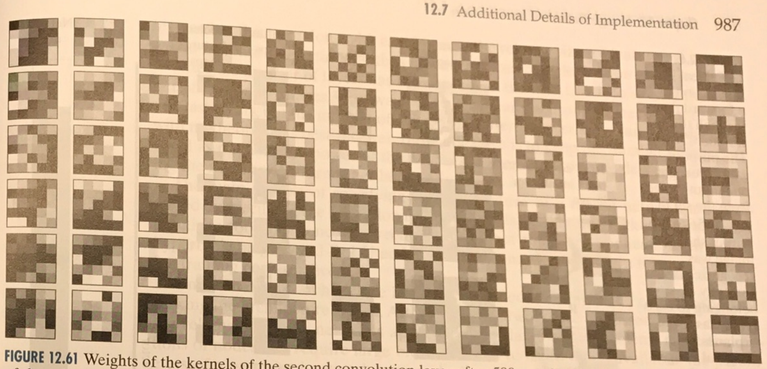


FIGURE 12.61 Weights of the kernels of the second convolution layer after 500 epochs of training. The interpretation of these kernels is the same as in Fig. 12.54.

12.7 SOME ADDITIONAL DETAILS OF IMPLEMENTATION

We mentioned in the previous section that neural (including convolutional) nets have the ability to learn features directly from training data, thus reducing the need for "engineered" features. While this is a significant advantage, it does not imply that the design of a neural network is free of human input. On the contrary, designing complex neural networks requires significant skill and experimentation.

In the last two sections, our focus was on the development of fundamental concepts in neural nets, with an emphasis on the derivation of backpropagation for both fully connected and convolutional nets. Backpropagation is the backbone of neural net design, but there are other important considerations that influence how well a neural net learns, and then generalizes to patterns it has not seen before. In this section, we discuss briefly some important aspects in the design of fully connected and convolutional neural networks.

One of the first questions when designing a neural net architecture is how many layers to specify for the network. Theoretically, the *universality approximation theorem* (Cybenco [1989]) tells us that, under mild conditions, arbitrarily complex decision functions can be *approximated* by a continuous feedforward neural network with a single hidden layer. Although the theorem does not tell us how to compute the parameters of that single hidden layer, it does indicate that structurally simple neural nets can be very powerful. You have seen this in some of the examples in the last two sections. Experimental evidence suggests that deep neural nets (i.e., networks with two or more hidden layers) are better than a single hidden layer network at learning abstract representations, which typically is the main point of learning. There is no such thing as an algorithm to determine the "optimum" number of layers to use in a neural network. Therefore, specifying the number of layers generally

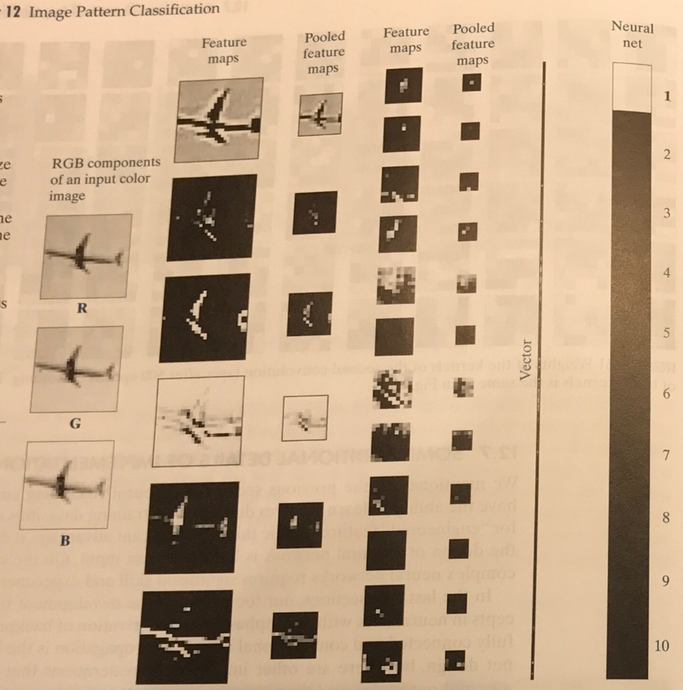


FIGURE 12.62 Graphical illustration of a forward pass through the trained CNN. The purpose was to recognize one input image from the set in Fig. 12.56. As the output shows, the image was recognized correctly as belonging to class 1, the class of airplanes. (Original image courtesy of Pearson Education.)

is determined by a combination of experience and experimentation. "Starting small" is a logical approach to this problem. The more layers a network has, the higher the probability that backpropagation will run into problems such as so-called *vanishing gradients,* where gradient values are so small that gradient descent ceases to be effective. In convolutional networks, we have the added issue that the size of the inputs decreases as the images propagate through the network. There are two causes for this. The first is a natural size reduction caused by convolution itself, with the amount of reduction being proportional to the size of the receptive fields. One solution is to use *padding* prior to performing convolution operations, as we discussed in Section 3.4. The second (and most significant) cause of size reduction is pooling. The minimum pooling neighborhood is of size 2×2, which reduces the size of feature maps by three-quarters at each layer. A solution that helps is to *upsample* the input

images, but this must done with care because the relative sizes of features of interest would increase proportionally, thus influencing the size selected for receptive fields.

After the number of layers has been specified, the next task is to specify the number of neurons per layer. We always know how many neurons are needed in the first and last layers, but the number of neurons for internal layer is also an open question with no theoretical "best" answer. If the objective is to keep the number of layers as small as possible, the power of the network is increased to some degree by increasing the number of neurons per layer.

The main aspects of specifying the architecture of a neural network are completed by specifying the activation function. In this chapter, we worked with sigmoid functions for consistency between examples, but there are applications in which hyperbolic tangent and ReLU activation functions are superior in terms of improving training performance.

Once a network architecture has been specified, training is the central aspect of making the architecture useful. Although the networks we discussed in this chapter are relatively simple, networks applied to very large-scale problems can have millions of nodes and require large blocks of time to train. When available, the parameters of a *pretrained* network are an ideal starting point for further training, or for validating recognition performance. Another central theme in training neural nets is the use of GPUs to accelerate matrix operations.

An issue often encountered in training is *over-fitting*, in which recognition of the training set is acceptable, but the recognition rate on samples not used for training is much lower. That is, the net is not able to *generalize* what it learned and apply it to inputs it has not encountered before. When additional training data is not available, the most common approach is to artificially enlarge the training set using transformations such as geometric distortions and intensity variations. The transformations are carried out while preserving the class membership of the transformed patterns. Another major approach is to use *dropout*, a technique that randomly drops nodes with their connections from a neural network during training. The idea is to change the architecture slightly to prevent the net from adapting too much to a fixed set of parameters (see Srivastava et al. [2014]).

In addition to computational speed, another important aspect of training is efficiency. Simple things, such as shuffling the input patterns at the beginning of each training epoch can reduce or eliminate the possibility of "cycling," in which parameter values repeat at regular intervals. *Stochastic gradient descent* is another important training refinement in which, instead of using the entire training set, samples are selected randomly and input into the network. You can think of this as dividing the training set into *mini-batches*, and then choosing a single sample from each mini-batch. This approach often results in speedier convergence during training.

In addition to the above topics, a paper by LeCun et al. [2012] is an excellent overview of the types of considerations introduced in the preceding discussion. In fact, the breath spanned by these topics is extensive enough to be the subject of an entire book (see Montavon et al. [2012]). The neural net architectures we discussed were by necessity limited in scope. You can get a good idea of the practical requirements of implementing practical networks by reading a paper by Krizhevsky, Sutskever, and Hinton [2012], which summarizes the design and implementation of a large-scale, deep convolutional neural network. There are a multitude of designs that have

been implemented over the past decade, including commercial and free implementations. A quick internet search will reveal a multitude of available architectures.

Summary, References, and Further Reading

Background material for Sections 12.1 through 12.4 are the books by Theodoridis and Koutroumbas [2006], by Duda, Hart, and Stork [2001], and by Tou and Gonzalez [1974]. For additional reading on the material on matching shape numbers see Bribiesca and Guzman [1980]. On string matching, see Sze and Yang [1981]. A significant portion of this chapter was devoted to neural networks. This is a reflection of the fact that neural nets, and in particular convolutional neural nets, have made significant strides in the past decade in solving image pattern classifications problems. As in the rest of the book, our presentation of this topic focused on fundamentals, but the topics covered were thoroughly developed. What you have learned in this chapter is a solid foundation for much of the work being conducted in this area. As we mentioned earlier, the literature on neural nets is vast, and quickly growing. As a starting point, a basic book by Nielsen [2015] provides an excellent introduction to the topic. The more advanced book by Goodfellow, Bengio, and Courville [2016] provides more depth into the mathematical underpinning of neural nets. Two classic papers worth reading are by Rumelhart, Hinton, and Williams [1986], and by LeCun, Bengio, and Haffner [1998]. The LeNet architecture we discussed in Section 12.6 was introduced in the latter reference, and it is still a foundation for image pattern classification. A recent survey article by LeCun, Bengio, and Hinton [2015] gives an interesting perspective on the scope of applicability of neural nets in general. The paper by Krizhevsky, Sutskever, and Hinton [2012] was one of the most important catalysts leading to the significant increase in the present interest on convolutional networks, and on their applicability to image pattern classification. This paper is also a good overview of the details and techniques involved in implementing a large-scale convolutional neural network. For details on the software aspects of many of the examples in this chapter, see Gonzalez, Woods, and Eddins [2009].

Problems

Solutions to the problems marked with an asterisk (*) are in the DIP4E Student Support Package (consult the book website: www.ImageProcessingPlace.com).

12.1 Do the following:

(a)* Compute the decision functions of a minimum distance classifier for the patterns in Fig. 12.10. You may obtain the required mean vectors by (careful) inspection.

(b) Sketch the decision boundary implemented by the decision functions in (a).

12.2* Show that Eqs. (12-3) and (12-4) perform the same function in terms of pattern classification.

12.3 Show that the boundary given by Eq. (12-8) is the perpendicular bisector of the line joining the n-dimensional points $\mathbf{m}_i$ and $\mathbf{m}_j$.

12.4* Show how the minimum distance classifier discussed in connection with Fig. 12.11 could be implemented by using N_c resistor banks (N_c is the number of classes), a summing junction at each bank (for summing currents), and a maximum selector capable of selecting the maximum value of N_c decision functions in order to determine the class membership of a given input.

12.5* Show that the correlation coefficient of Eq. (12-10) has values in the range $[-1, 1]$. (*Hint*: Express γ in vector form.)

12.6 Show that the distance measure $D(a,b)$ in Eq. (12-12) satisfies the properties in Eq. (12-13).

12.7* Show that $\beta = \max\left(|a|,|b|\right) - \alpha$ in Eq. (12-14) is 0 if and only if a and b are identical strings.

12.8 Carry out the manual computations that resulted in the mean vector and covariance matrices in Example 12.5.

12.9* The following pattern classes have Gaussian probability density functions:

$$c_1 : \{(0,0)^T, (2,0)^T, (2,2)^T, (0,2)^T\}$$
$$c_2 : \{(4,4)^T, (6,4)^T, (6,6)^T, (4,6)^T\}$$

(a) Assume that $P(c_1) = P(c_2) = 1/2$ and obtain the equation of the Bayes decision boundary between these two classes.

(b) Sketch the boundary.

12.10 Repeat Problem 12.9, but use the following pattern classes:

$$c_1 : \{(-1,0)^T, (0,-1)^T, (1,0)^T, (0,1)^T\}$$
$$c_2 : \{(-2,0)^T, (0,-2)^T, (2,0)^T, (0,2)^T\}$$

Note that the classes are not linearly separable.

12.11 With reference to the results in Table 12.1, compute the overall correct recognition rate for the patterns of the training set. Repeat for the patterns of the test set.

12.12* We derived the Bayes decision functions

$$d_j(\mathbf{x}) = p(\mathbf{x}/c_j)P(c_j),\ j = 1,2,\ldots,N_c$$

using a 0-1 loss function. Prove that these decision functions minimize the probability of error. (*Hint:* The probability of error $p(e)$ is $1 - p(c)$, where $p(c)$ is the probability of being correct. For a pattern vector $\mathbf{x}$ belonging to class c_i, $p(c/\mathbf{x}) = p(c_i/\mathbf{x})$. Find $p(c)$ and show that $p(c)$ is maximum [$p(e)$ is minimum] when $p(\mathbf{x}/c_i)P(c_i)$ is maximum.)

12.13 Finish the computations started in Example 12.7.

12.14* The perceptron algorithm given in Eqs. (12-44) through (12-46) can be expressed in a more concise form by multiplying the patterns of class c_2 by -1, in which case the correction steps in the algorithm become $\mathbf{w}(k+1) = \mathbf{w}(k)$, if $\mathbf{w}^T(k)\mathbf{y}(k) > 0$, and $\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha\mathbf{y}(k)$ otherwise, where we use $\mathbf{y}$ instead of $\mathbf{x}$ to make it clear that the patterns of class c_2 were multiplied by -1. This is one of several perceptron algorithm formulations that can be derived starting from the general gradient descent equation

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \alpha\left[\frac{\partial J(\mathbf{w},\mathbf{y})}{\partial \mathbf{w}}\right]_{\mathbf{w}=\mathbf{w}(k)}$$

where $\alpha > 0$, $J(\mathbf{w},\mathbf{y})$ is a criterion function, and the partial derivative is evaluated at $\mathbf{w} = \mathbf{w}(k)$. Show that the perceptron algorithm in the problem statement can be obtained from this general gradient descent procedure by using the criterion function

$$J(\mathbf{w},\mathbf{y}) = \frac{1}{2}\left(|\mathbf{w}^T\mathbf{y}| - \mathbf{w}^T\mathbf{y}\right)$$

(*Hint:* The partial derivative of $\mathbf{w}^T\mathbf{y}$ with respect to $\mathbf{w}$ is $\mathbf{y}$.)

12.15* Prove that the perceptron training algorithm given in Eqs. (12-44) through (12-46) converges in a finite number of steps if the training pattern sets are linearly separable. [*Hint:* Multiply the patterns of class c_2 by -1 and consider a non-negative threshold, T_0 so that the perceptron training algorithm (with $\alpha = 1$) is expressed in the form $\mathbf{w}(k+1) = \mathbf{w}(k)$, if $\mathbf{w}^T(k)\mathbf{y}(k) > T_0$, and $\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha\mathbf{y}(k)$ otherwise. You may need to use the Cauchy-Schwartz inequality: $\|\mathbf{a}\|^2\|\mathbf{b}\|^2 \geq (\mathbf{a}^T\mathbf{b})^2$.]

12.16 Derive equations of the derivatives of the following activation functions:

(a) The sigmoid activation function in Fig. 12.30(a).

(b) The hyperbolic tangent activation function in Fig. 12.30(b).

(c)* The ReLU activation function in Fig. 12.30(c).

12.17* Specify the structure, weights, and bias(es) of the smallest neural network capable of performing *exactly* the same function as a minimum distance classifier for two pattern classes in n-dimensional space. You may assume that the classes are tightly grouped and are linearly separable.

12.18 What is the decision boundary implemented by a neural network with n inputs, a single output neuron, and no hidden layers? Explain.

12.19 Specify the structure, weights, and bias of a neural network capable of performing exactly the same function as a Bayes classifier for two pattern classes in n-dimensional space. The classes are Gaussian with different means but equal covariance matrices.

12.20 Answer the following:

(a)* Under what conditions are the neural networks in Problems 12.17 and 12.19 identical?

(b) Suppose you specify a neural net architecture identical to the one in Problem 12.17. Would training by backpropagation yield the same