Rafael C. Gonzalez

Deep Convolutional Neural Networks

Neural networks are a subset of the field of artificial intelligence (AI). The predominant types of neural networks used for multidimensional signal processing are *deep convolutional neural networks* (CNNs). The term *deep* refers generically to networks having from a "few" to several dozen or more convolution layers, and *deep learning* refers to methodologies for training these systems to automatically learn their functional parameters using data representative of a specific problem domain of interest. CNNs are currently being used in a broad spectrum of application areas, all of which share the common objective of being able to automatically learn features from (typically massive) data bases and to generalize their responses to circumstances not encountered during the learning phase. Ultimately, the learned features can be used for tasks such as classifying the types of signals the CNN is expected to process. The purpose of this "Lecture Notes" article is twofold: 1) to introduce the fundamental architecture of CNNs and 2) to illustrate, via a computational example, how CNNs are trained and used in practice to solve a specific class of problems.

Relevance

After decades of languishing in research laboratories, AI has recently experienced an explosion in worldwide interest as a strategic tool in industry, government, and research institutions. This interest is based on the fact that AI makes it possible for computers to learn from experience, generalize their behavior, and perform tasks that one normally associates with human intelligence. Some applications of AI are well known to the general public, such as computers that beat grand masters at chess, recognize fingerprints, and interpret verbal commands. Other applications are less well known, such as fraud detection, searching for patterns in large amounts of data, and controlling complex industrial processes. As varied as they are, however, all of these applications are based on the same concepts from deep learning. Of particular interest in two-dimensional (2-D) signal processing is automatic recognition of the contents of digital images using deep learning, which is currently being applied with unprecedented success in fields ranging from biometrics, such as face and retinal identification, to visual quality inspection, medical diagnoses, and autonomous vehicle navigation.

The purpose of this "Lecture Notes" article is twofold: 1) to introduce the fundamental architecture of CNNs and 2) to illustrate, via a computational example, how CNNs are trained and used in practice to solve a specific class of problems.

Prerequisites

The only prerequisites for understanding this article are calculus (in particular, differentiation and the chain rule) and linear algebra, both at the undergraduate level.

Background and problem statement

Interest in using computers to perform automated image recognition tasks dates back more than half a century. During the mid 1950s and early 1960s, a class of so-called learning machines [1] caused a great deal of excitement in the field of machine learning. The reason was the development of mathematical proofs showing that basic computing units, called *perceptrons*, when trained with linearly separable data sets, would converge to a solution in a finite number of iterative steps. The solution took the form of coefficients of hyperplanes that were capable of correctly separating these data classes in feature hyperspace. Unfortunately, the basic perceptron was inadequate for tasks of practical significance. Subsequent attempts to extend the power of perceptrons by assembling multiple layers of these devices lacked effective training algorithms, such as those that had created interest in the perceptron itself [2]. This discouraging state of the art changed with the development in 1986 of *backpropagation*, a method for training neural networks

Digital Object Identifier 10.1109/MSP.2018.2842646
Date of publication: 13 November 2018

composed of layers of perceptron-like units [3]. Backpropagation was first applied to 2-D signals in 1989 in the context of what we now refer to as *deep CNNs* [4]. Similar efforts followed at a relatively low level for the next two decades, but it was not until 2012, when publication of the results of the 2012 ImageNet Challenge demonstrated the power of deep CNNs, that these neural nets began to be used widely in image pattern recognition and other imaging applications [5], [6]. Today, CNNs are the approach of choice for addressing complex image recognition tasks and other important fields, which will be mentioned shortly.

Pattern recognition by machine involves the following four basic stages:

1) acquisition
2) preprocessing
3) feature extraction
4) classification.

Acquisition generates the raw input patterns (e.g., digital images); preprocessing deals with tasks such as noise reduction and geometric corrections; feature extraction deals with computing attributes that are fundamental in differentiating one class of patterns from another; and classification is the process that assigns a given input pattern to one of several predefined classes. Feature extraction usually is the most difficult problem to solve, with extensive engineering often being required to define and test a suitable set of features for a given application. CNNs offer an alternative approach that automates the learning of features by utilizing large databases of samples, called *training sets*, that are representative of an application domain of interest.

The problem addressed in this tutorial is to define a CNN-based strategy for extracting features automatically from a large training database and to use those features for accurately recognizing images from both the training database and also from an independent set of test images. This type of problem is by far the predominant application of CNNs, but it is not their only use. CNNs are currently being applied successfully in a number of other areas that include speech recognition, semantic image segmentation, and natural language processing [8]. In each case, the specifics of how CNNs are structured may vary, but their principles of operation are the same as those discussed in this article.

Solution

We approach the solution to the problem stated in the previous section by using a deep modular CNN architecture consisting of layers of convolution, activation, and pooling. The output of the CNN is then fed into a deep, *fully connected neural network* (FCN), whose purpose is to map a set of 2-D features into a class label for each input image. Central to this approach is the ability to use sample training data to learn the operational parameters of each network layer. For this, we use backpropagation as a tool for iteratively adjusting the network weights (also referred to as *coefficients*, *parameters*, and *hyperparameteres*) based on cycling through the training data. Finally, we demonstrate the effectiveness of the solution by training the CNN/FCN system using a large database of handwritten numeric characters and then testing it with a set of images not used in the training phase. As we show in the "A Computational Example" section, the recognition accuracy achieved by the system on the images of both data sets exceeded 99%.

The fundamental operation performed in each stage of a CNN is convolution, from which these neural nets derive their name.

Deep CNNs

Figure 1 shows the basic components of one stage of a CNN. In practice, a CNN can have tens of such stages, interconnected in series. In addition to the number of stages, CNN architectures differ in how the elements of each stage are defined and used, but the basic structure in Figure 1 is fundamental to all of them.

As the figure shows, one stage of a CNN is composed in general of three volumes, consisting, respectively, of *input maps*, *feature maps*, and *pooled feature maps* (or *pooled maps*, for short). Pooled maps are not always used in every stage and, in some applications, not at all. All maps are 2-D arrays whose size generally varies from volume to volume, but all maps within a volume are of the same size. If the input to the CNN is an RGB color image, the input volume will consist of three maps—the red, green, and blue component images, or channels, of the RGB image. The term *input maps volume* comes from the fact that the inputs have height and width (the spatial dimensions of each map) as well as depth, equal to the number of maps in a volume. In the context of our discussion, the input volume to the first stage consists in general of the channels of multispectral images; the input volumes to all other stages are the pooled maps (or feature maps for stages with no pooling) from the previous stage. When present, the number of pooled maps in a stage is equal to the number of feature maps.

The fundamental operation performed in each stage of a CNN is convolution, from which these neural nets derive their name. Although convolution is a ubiquitous operation in signal processing, it is not always explicitly stated that the type of convolution performed in CNNs is, in general, volume convolution, with the restriction that there is no displacement of the convolution kernel volume (also referred to as a *filter*) in the depth dimension. Figure 1 illustrates this concept, in which a kernel volume, shown in yellow, consists of three individual 2-D kernels. It is evident from this figure that the depth of each kernel volume in any stage is always equal to the depth of the input volume to that stage. Convolution is performed between a different 2-D kernel and its corresponding 2-D input map. Because there is no displacement in the depth dimension, a volume convolution in this case is simply the sum of the individual 2-D convolutions. To understand how a CNN works, it helps to focus attention on the result of volume convolution at one pair of spatial coordinates, (x, y).

Let $w_{m,n,k}$ denote the weights of a 2-D kernel associated with the kth map in the input volume, where m and n are variables that index over the kernel height and width. The convolution between this kernel and the kth map, at any specific spatial location, (x, y), of

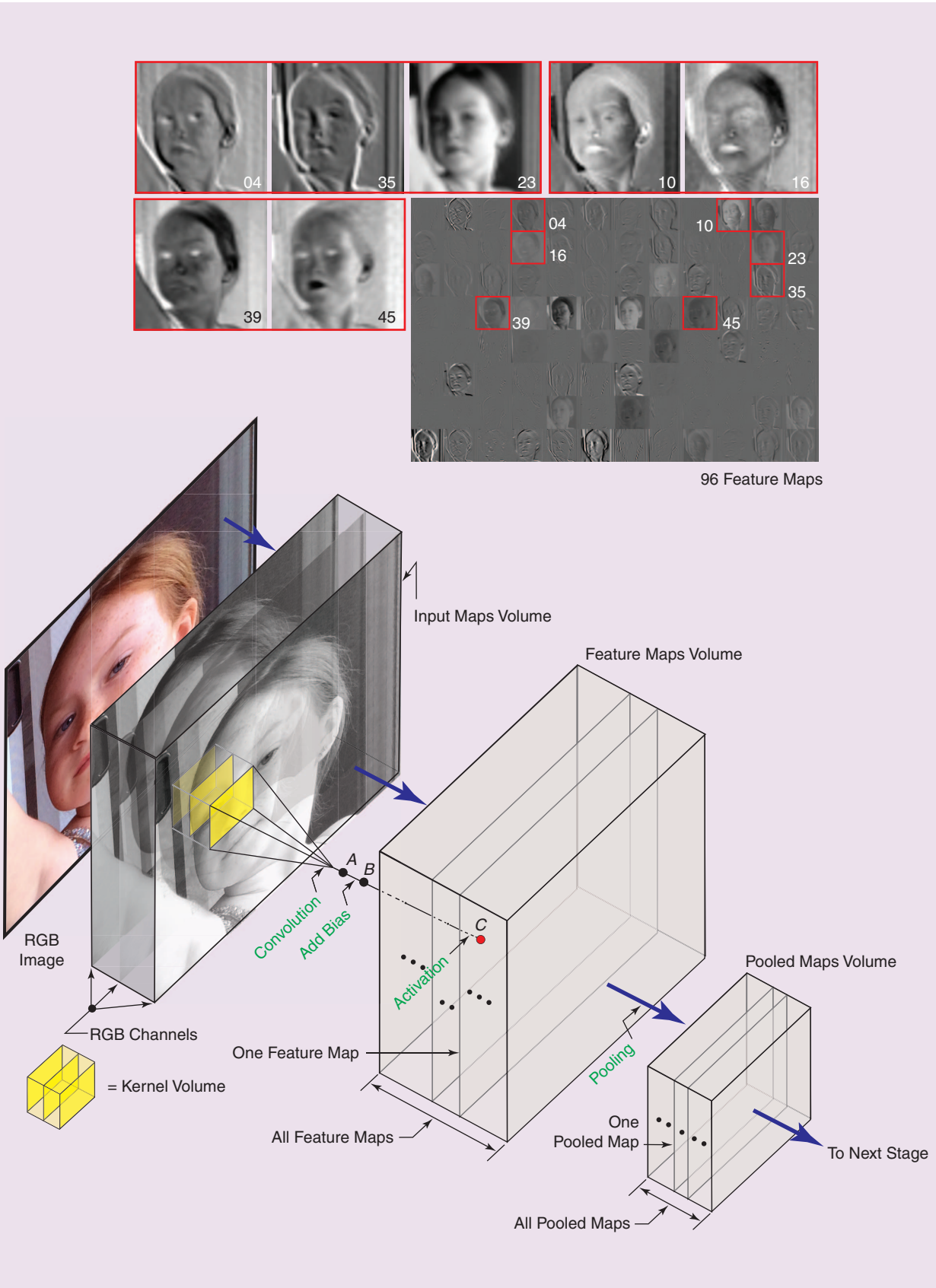

FIGURE 1. The components of one stage of a CNN, consisting of an input maps volume, a feature maps volume, and an optional pooled maps volume. The maps in the input volume correspond to the three channels of the RGB image shown. The stage has 96 feature maps and 96 pooled maps. The highlighted feature maps, displayed as images and identified numerically, illustrate the types of features that a CNN is capable of extracting from an input image.

the map, is the sum of products of the weights of the kernel and the elements of the map that are spatially coincident with the kernel. To obtain a volume convolution, the sum of products operation is performed between each corresponding 2-D kernel and its map at that same spatial location. Each sum of products is a scalar, and the volume convolution at that point is the sum of the K resulting scalars, where K is the depth of the input volume. To write this in equation form would require K 2-D summations. However, for reasons that will be explained in the next section, we can redefine the indices and write the K summations as one:

$$\text{conv}_{x,y} = \sum_i w_i v_i, \tag{1}$$

where the ws are kernel weights, the vs are values of the spatially corresponding elements in the input maps, and $\text{conv}_{x,y}$ is the result of volume convolution at the same spatial coordinates, (x, y), for all maps of the input volume. Equation (1) gives the result at point A in Figure 1. The result at point B is obtained by adding a scalar bias, b, to (1)

$$z_{x,y} = \sum_i w_i v_i + b. \tag{2}$$

We discuss the nature of this bias in the next section.

The result at point C is obtained by passing scalar $z_{x,y}$ through a nonlinearity called an *activation function*, h

$$a_{x,y} = h(z_{x,y}). \tag{3}$$

Activation functions used in practice include sigmoids $h(z) = 1/(1 + \exp(-z))$, hyperbolic tangents $h(z) = \tanh(z)$, and so-called rectified linear units (ReLUs) $h(z) = \max(0, z)$. The resulting $a_{x,y}$, called an *activation value,* becomes the value of the feature map at location (x, y), as illustrated by the point labeled C in Figure 1. A complete feature map, also referred to as an *activation map*, is generated by performing the three operations just explained at all spatial locations of the input maps. Each feature map has one kernel volume and one bias associated with it. The objective is to use training data to learn the weights of the kernel volume and bias of each feature map. We explain in the following two sections how these coefficients are learned, and give a detailed computational example of a CNN application.

Figure 1 also illustrates the types of features that volume convolution is able to extract. The input to the CNN stage in Figure 1 was an RGB image of size 277×277 pixels, which resulted in an input volume of depth three, corresponding to the red, green, and blue channels of the RGB image. We used the image of a human subject as the input so that the resulting feature maps would be easier to interpret visually. The feature maps volume in this case was specified to have 96 feature maps, each obtained by filtering the maps of the input volume with a different kernel volume of size $11 \times 11 \times 3$. Thus, there are 96 kernel volumes of depth three, for a total of $3 \times 96 = 288$ 2-D convolution kernels of size 11×11 in this CNN stage. The 96 feature maps resulting from the input image are shown as images in the upper right of Figure 1 as an 8×12 montage. The feature maps shown in enlarged detail are numbered and grouped to illustrate the variety of complementary features that can result from volume convolution. The first group shows three feature maps. Two of them (4 and 35) emphasize edge content, and the third (23) is a blurred version of the input. The second group has two maps (10 and 16) that capture complementary shades of gray (note the difference in the hair intensity, for example). In the third group, feature map 39 emphasizes the subject's eyes and dress, both of which are blue in the input RGB image. Map 45 also emphasizes blue, but it also emphasizes areas that correspond to red tones in the RGB image, such as the subject's lips, hair, and skin. These two feature maps are more sensitive to color content than the maps in the other two groups. Subsequent stages would operate on these feature maps to extract further abstractions from the data, as we illustrate later in the "A Computational Example" section. The weights of the convolution kernel volumes used to generate the 96 feature maps came from AlexNet, a CNN trained using more than 1 million images belonging to 1,000 object categories [5]. The system had never "seen" the image we used in Figure 1.

The pooling, or subsampling, shown in Figure 1 is motivated by studies that suggest that the brains of mammals perform an analogous operation during visual cognition. A pooled map is simply a feature map of lower resolution. A typical pooling method is to replace the values of every neighborhood of size, say, 2×2, in the feature maps by the average of the values in the neighborhood. Using a neighborhood of size 2×2 results in pooled maps of size one-half in each spatial dimension of the size of the feature maps. Thus, a consequence of pooling is significant data reduction, which helps speed up processing. However, a major disadvantage is that map size also decreases significantly every time pooling is performed. Even with neighborhoods of size 2×2 the reduction by half in each spatial dimension quickly becomes an issue when the number of layers is large with respect to the size of the input images. This is one of the reasons why pooling is used only sporadically in large CNN systems. As with activation functions, the type of pooling used also plays a role in defining the architecture of a CNN. In addition to *neighborhood averaging*, two additional pooling methods used in practice are *max pooling*, which replaces the values in a neighborhood by the maximum value of its elements, and *L2 pooling*, in which the pooled value in a neighborhood is the square root of the sum of their values squared. Max pooling has been demonstrated to be particularly effective in classifying large image databases, and it has the added advantage of simplicity and speed. As noted previously, when pooling is used in a layer, each pooled map is generated from only one feature map, so the number of feature and pooled maps is the same.

The basic architecture of each stage of a CNN is defined by specifying the number of feature maps and by whether or not pooling is used in that stage. Also specified are kernel and pooling sizes, and the *convolution stride*, defined as the number of increments of displacement of the kernel between convolution operations. For example, a stride of two means that convolution is performed at every other spatial location in the input

maps. The number of 2-D convolution kernels needed in each stage is equal to the depth of the input volume multiplied by the number of feature maps. The spatial dimensions of all kernels in a stage are the same and are specified as part of the definition of a CNN stage. Generally, the same type of activation is used in all stages of a CNN. This is true also of the size and type of pooling method used when pooling is defined for one or more stages of the network.

There are two major ways in which CNNs are structured: A fully convolutional network (not to be confused with a fully connected network) consists exclusively of stages of the form described in Figure 1, connected in series. The major application of fully convolutional architectures is image segmentation in which the objective is labeling each individual pixel in an input image. Because map size decreases as the number of stages increases, additional processing, such as upsampling, is used so that the output maps are of the same size as the input images. In fact, fully convolutional nets can be connected "end to end" so that map size is first allowed to decrease as a result of convolution and then are run in a reverse process through an identical network whose maps increase from stage to stage using "backward" convolution. The final output is an image of the same size as the input, but in which pixels have been labeled and grouped into regions [8].

The second major way in which CNNs are used is for image classification which, as noted previously, is by far the widest use of CNNs. In this application, the output maps in the last stage of a CNN are fed into an FCN whose function is to classify its input into one of a predetermined number of classes. Because the output volume of a CNN consists of 2-D maps and, as we will show in the next section, the inputs to FCNs are vectors, the interface between a CNN and an FCN is a simple stage that converts 2-D arrays to vectors. A discussion of how all of this is accomplished and applied to solve a specific problem is the subject of the section "A Computational Example."

The objective of training a CNN/FCN network is to determine the weights and biases of convolution volumes in the former, and of the neuron weights and biases in the latter, that solve a given problem.

Deep FCNs

A single perceptron is a computational unit that performs a sum-of-products operation, $z = \sum_{i=1}^{n} w_i x_i + w_{n+1}$, between a set of weights, $w_1, w_2, \ldots, w_n, w_{n+1}$, and a set of input scalar pattern features, $x_1, x_2, \ldots x_n$. A vector formed from these features is referred to as a *pattern* (or *feature*) vector. Setting $z = 0$ gives the equation of an n-dimensional hyperplane, where coefficient w_{n+1} is a bias that offsets the hyperplane from the origin of the corresponding n-dimensional Euclidean space. In the "classic" perceptron, the output of the sum-of-products computation is fed into a hard threshold, h, to produce an activation value, $a = h(z)$, with a binary output denoted typically by $[+1, -1]$. Then, if $a = 1$, an input pattern is assigned by the single perceptron to one class, and, if $a = -1$, the pattern is assigned to another. Neural networks are composed of perceptrons in which the activation function is changed from a hard threshold to a smoother function, such as a sigmoid, hyperbolic tangent, or ReLU function, as defined in the previous section. The resulting unit is referred to as an *artificial neuron* because of postulated similarities between its response and the way neurons in the brains of mammals are believed to function.

Figure 2 is a schematic of a deep FCN consisting of layers of artificial neurons in which the output of every neuron in a layer is connected to the input of every neuron in the next layer, hence the term *fully connected*. The input layer is formed from the components of a pattern vector, $x_1, x_2, \ldots, x_n$, and the number of neurons in the output layer is equal to the number of pattern classes in a given application. The input and output layers are visible because we can observe the values of their outputs. All other layers in a neural net are *hidden layers*. Note that CNNs are not fully connected, in the sense that each element of a map in one layer is not connected to every element of maps in the following layer.

The objective of training a CNN/FCN network is to determine the weights and biases of convolution volumes in the former, and of the neuron weights and biases in the latter, that solve a given problem. As noted in the "Background and Problem Statement" section, these parameters are estimated using backpropagation, a methodology for iteratively adjusting the coefficients based on values of the error observed at the output neurons of the FCN.

The computation performed by the zoomed neuron in Figure 2 is

$$z_i(\ell) = \sum_{j=1}^{n_{\ell-1}} w_{ij}(\ell) a_j(\ell-1) + b_i(\ell), \quad (4)$$

where $w_{ij}(\ell)$ is the weight of the ith neuron in layer ℓ that associates that neuron with the output of the jth neuron in layer $\ell-1$; $a_j(\ell-1)$ is the output of the jth neuron in layer $\ell-1$; $b_i(\ell)$ is the bias of the ith neuron in layer ℓ; and $n_{\ell-1}$ is the number of neurons in layer $\ell-1$. The output of the ith neuron is obtained by passing $z_i(\ell)$ through a nonlinearity, h, of the form discussed in the previous section:

$$a_i(\ell) = h(z_i(\ell)). \quad (5)$$

These two simple expressions completely characterize the behavior of a neuron in any layer of an FCN. Basically, these equations indicate that the inputs to a neuron in any layer of an FCN are the outputs of all neurons in the previous layer and that the output of that neuron is the sum of products of the neuron weights and its inputs, to which we add a scalar value, and then pass the total sum through a nonlinearity. The important thing to note in (4) and (5) is that they are identical in form to (2) and (3), indicating that CNNs and FCNs perform the same types of neural computations. The ultimate result of this similarity is that training a CNN and an FCN follows the same computational rules, with allowances being made for the fact that CNNs operate on volumes, while FCNs work with vectors.

Training of an FCN begins by assigning small random values to all weights and

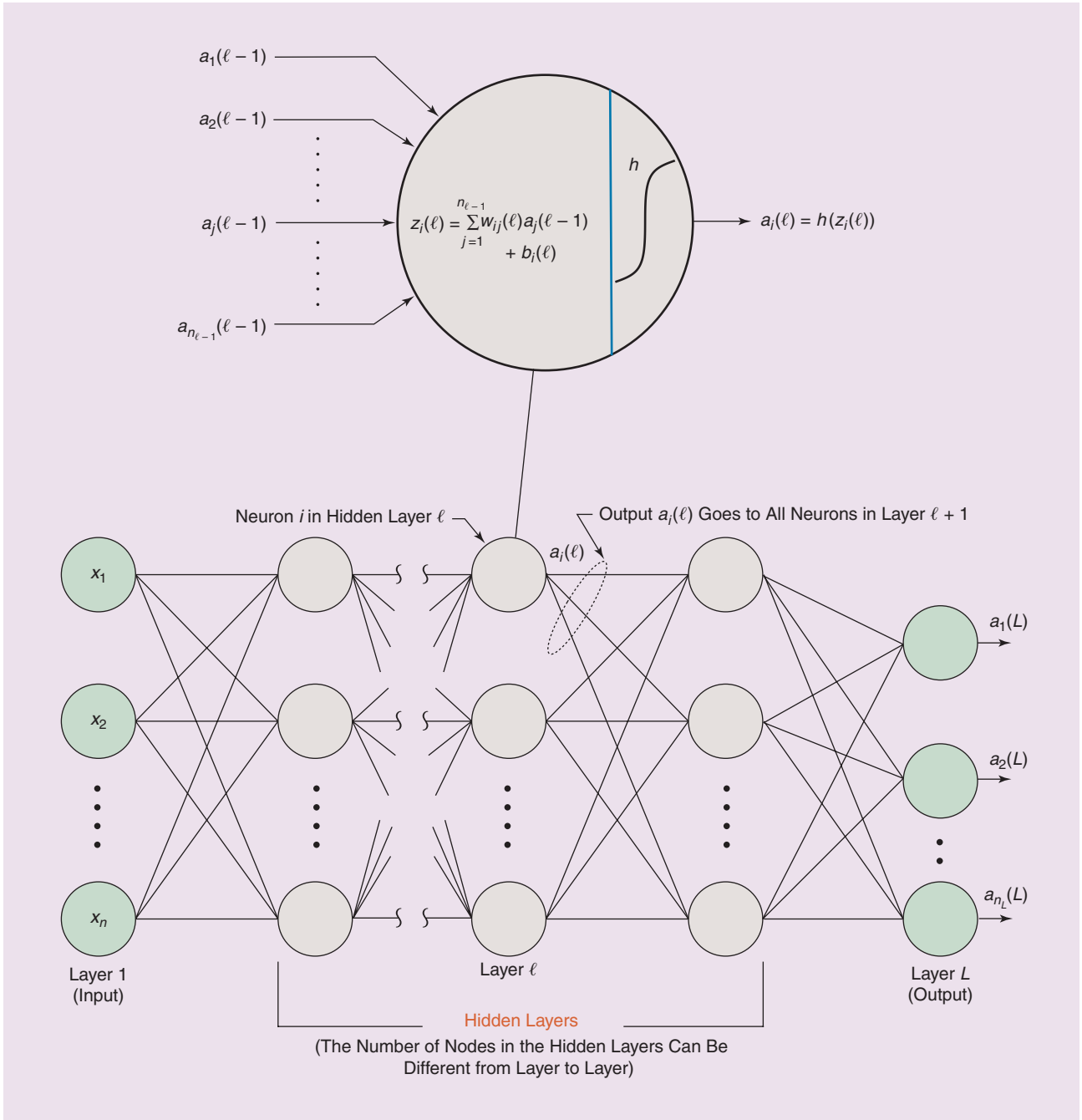

FIGURE 2. A schematic of a fully connected neural network. The zoomed section shows the computations performed by each neuron in the network. The activation function, h, shown is in the shape of a sigmoid.

biases. Because we know that $a_j(1) = x_j$, we can use (4) and (5) to compute $z_j(\ell)$ and $a_j(\ell)$ for all layers in the network, past the first. Although it is not shown in the diagram, we also compute $h'(z_i(\ell))$ for use later in backpropagation. Propagating a pattern vector through a neural net to its output is called *feedforward*, and training consists of feedforward and backpropagation passes through the network, periodically adjusting the weights and biases between such passes.

Measuring performance during training requires an error, or cost, function. The function used most frequently for this purpose is the mean-squared error (MSE) between actual and desired outputs:

$$E = \frac{1}{2}\sum_{j=1}^{n_L}(r_j - a_j(L))^2, \qquad (6)$$

where $a_j(L)$ is the activation value of the jth neuron in the output layer of the FCN. During training, we let $r_j = 1$ if the pattern being processed belongs to the jth class and $r_j = 0$ if it does not. Thus, if a pattern belongs to the kth class, we want the response of the kth output neuron, $a_k(L)$, to be 1 and the response of all other output neurons to be 0. When this occurs, the error is zero and no adjustments

are made to the weights because the input vector was classified correctly.

The objective of training is to adjust all the weights and biases in the network when a classification mistake is made, so that the error at the output is minimized. This is done using gradient descent for the weights and biases

$$w_{ij}(\ell) = w_{ij}(\ell) - \alpha \frac{\partial E}{\partial w_{ij}(\ell)} \tag{7}$$

and

$$b_i(\ell) = b_i(\ell) - \alpha \frac{\partial E}{\partial b_i(\ell)}, \tag{8}$$

where α is a scalar correction increment called the *learning rate constant*. Unfortunately, the change in the output error with respect to changes in the weights and biases in the hidden layers is not known. In a nutshell, backpropagation is a scheme that 1) propagates the error in the output, which is known, backward through all the hidden layers of the network and 2) uses the backpropagated error to express the two partials in (7) and (8) in terms of the activation function, the output error, and the current values of the weights and biases, all of which are known quantities at every layer in the network during training. A derivation of this important result is outside the scope of our discussion, but a sketch of the fundamental equations of backpropagation will help demonstrate the surprising simplicity of this method. The original derivation is given in [3], and is further illustrated and formulated in a more computationally effective matrix form, in [7].

Backpropagation is based on the following four results:

$$\frac{\partial E}{\partial w_{ij}(\ell)} = a_j(\ell - 1)\Delta_i(\ell) \tag{9}$$

and

$$\frac{\partial E}{\partial b_i(\ell)} = \Delta_i(\ell), \tag{10}$$

where

$$\Delta_j(\ell) = h'(z_j(\ell)) \sum_i w_{ij}(\ell + 1)\Delta_i(\ell + 1) \tag{11}$$

and

$$\Delta_j(L) = h'(z_j(L))[a_j(L) - r_j]. \tag{12}$$

Equations (9) and (10) are used to compute the gradients in (7) and (8), based on known or computable quantities. The fact that the quantities in (9) and (10) are known is established by (11) and (12). In the latter equation, $h'(z_j(L))$ and $a_j(L)$ are computed during feedforward, and r_j is given during training, so $\Delta_j(L)$ can be computed. But if we know this quantity, we can compute $\Delta_j(L-1)$ using (11) because all of its terms are known also during any training iteration. Another application of this equation gives $\Delta_j(L-2)$, and so on for all values of $\ell = L-1,\ L-2, \ldots, 2$. In other words, at any iterative step in training, we are able to compute all the quantities necessary to implement the gradient descent formulation given in (7) and (8), which seeks a minimum of the MSE in (6). Observe that we compute the terms necessary for gradient descent by proceeding backward from the output, hence the use of the term *backpropagation* to describe this method.

Training a CNN for image classification is performed in conjunction with training its attached FCN.

Using the preceding relatively simple equations, the procedure for training an FCN can be summarized as follows:

1) Initialize all weights and biases to small random values.
2) Using a pattern vector from the training set, perform a forward pass through the network and compute all values of $a_j(\ell)$ and $h'(z_j(\ell))$.
3) Compute the MSE using (6).
4) Compute $\Delta_j(L)$ using (12) and propagate it back through the network, using (11) to compute $\Delta_j(\ell)$ for $\ell = L-1, L-2, \ldots, 2$.
5) Update the weights and biases using (7)–(10).
6) Repeat steps 2–5 for all patterns of the training set. One pass through all training patterns constitutes one epoch of training. This procedure is repeated for a specified number of epochs, or until the MSE stabilizes to within a predefined range of acceptable variation.

Training a CNN for image classification is performed in conjunction with training its attached FCN. During feedforward, an image propagates through the CNN, resulting in a set of output maps in the last stage, as explained in Figure 1. The elements of these maps are vectorized and input into the FCN so that they propagate to the output of the fully connected net, at which point the MSE is computed, as described previously. The error delta, $\Delta_j(L)$, is backpropagated all the way to the input of the FCN. The vectorization applied on feedforward is then reversed into the 2-D format of the output maps. The reformatted quantities are the "deltas" of the CNN, which are then backpropagated to its input stage. The error deltas at each layer are computed during backpropagation through both networks, and these are then used to update the weights and biases of the CNN and FCN, using (7) and (8) for the latter, and their equivalents for the CNN [7]. Given the similarities between the computations performed by a CNN [(2) and (3)], and those performed by an FCN [(4) and (5)], the reader should not be surprised that the equations of backpropagation for the two networks are also similar. The fundamental difference between the equations for the two neural networks is that FCNs, which work with vectors, use multiplications, while CNNs, which work with 2-D arrays, use convolution.

As noted previously, the feedforward/backpropagation training procedure just explained is repeated for a specified number of epochs or until changes in the MSE stabilize to within a specified range of acceptable variation. After training, the CNN and FCN are completely specified by the learned weights and biases. When deployed for autonomous operation, the system classifies an unknown image into one of the classes on which the system was trained, by performing a feedforward pass and detecting which neuron at the output of the FCN yields the largest value.

A computational example

In this section, we illustrate how to train and test a CNN/FCN for image classification, using an image database that contains a training set of 60,000 grayscale images of handwritten numerals. The database also contains a set of 10,000 test images. Figure 3 shows the CNN and

FCN architectures we used. The layout is more detailed than in Figure 1 to simplify explanations. This network, which we explain below, was trained for 200 epochs using all 60,000 training images. The performance of the resulting trained system on the images of the training set was 99.4% correct classification. When subjected to the 10,000 test images, which the system had never "seen" before, the performance was 99.1%. These are impressive results, considering the simplicity of the architecture in Figure 3, and the fact that the inputs are handwritten characters that exhibit significant variability.

The input grayscale images are of size 28×28 pixels. The first stage of the CNN has six feature maps, and the second has 12. Both stages use pooling with 2×2 neighborhoods. The convolution kernels are of size 5×5 in both stages. The FCN has no hidden layers, consisting instead of only an input and an output layer. This means that the FCN is a linear classifier that implements hyperplane boundaries, as we noted previously in the discussion of perceptrons.

Because the inputs are grayscale images, the depth of the input volume to the first stage of the CNN is one, indicating that six 2-D kernels, one for each of the six feature maps, are needed in the first stage. The depth of the input volume to the second stage is six because there are six pooled maps at the output of the first stage. This means that 12 kernel volumes, each consisting of six 2-D kernels, are required to generate the 12 feature maps in the second stage, for a total of $6 \times 12 = 72$, 2-D convolution kernels in that stage. There is one bias per feature map, for a total of six biases in the first stage and 12 in the second.

For 2-D convolution without padding, we require that the 2-D kernels be completely contained in their respective maps during spatial translation. Because the input images are of size 28×28 pixels and the kernels are of size 5×5, this means that the feature maps in the first stage are of size 24×24 elements. Pooling reduces the size of these maps to 12×12 elements. These are the input maps to the second stage which, when convolved with kernels of size 5×5, result in feature maps of size 8×8. The

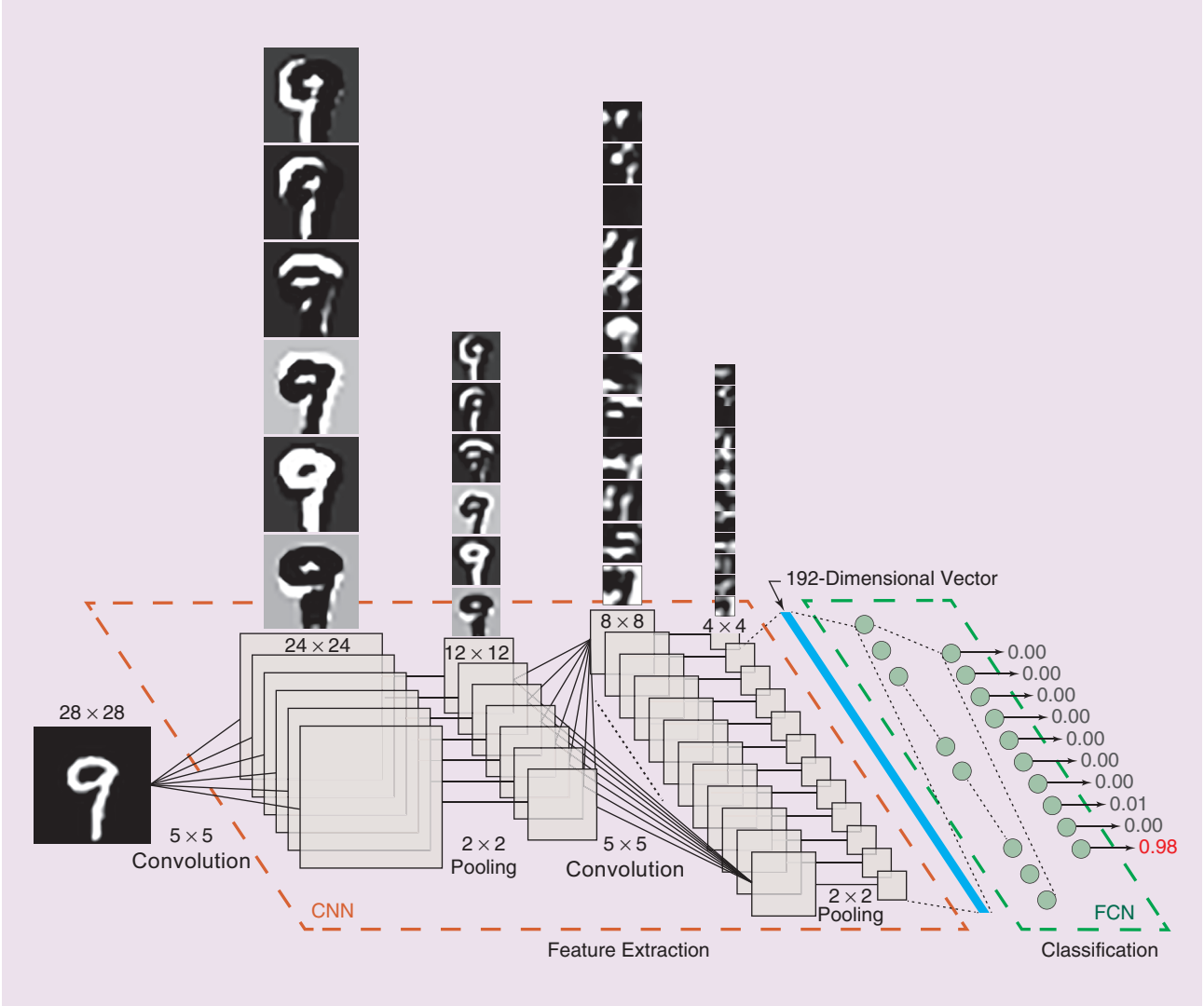

FIGURE 3. A CNN trained to extract features that are then used by an FCN to classify handwritten numerals. The input image shown is from the National Institute of Standards and Technology database. (A formatted version of this database is available for experimental work at yann.lecun.com/exdb/mnist.)

output maps in the second stage are obtained by pooling the feature maps in that stage, which results in 12 maps of size 4×4. These maps are then converted to vectors by linear indexing, which concatenates the elements of all the 2-D maps, column by column, into a one-dimensional string. When vectorized, these maps result in input vectors to the FCN that have $4 \times 4 \times 12 = 192$ elements. There are ten numeric classes, so the number of neurons in the output layer of the FCN is ten.

We illustrate the operations performed by our CNN/FCN neural net by following the flow of the image in Figure 3 from the input to the CNN to the output of the FCN. The weights and biases used in this example were obtained by training the CNN/FCN with the 60,000 images mentioned previously. Each feature map in the first stage of the CNN was generated by convolving a different 5×5 kernel with the input image. The resulting feature maps are shown as images above the feature maps volume in the first CNN stage. The feature maps in the first stage are of size 24×24 pixels, which we enlarged using bicubic interpolation to a size of 300×300 pixels, to make it easier to interpret them visually. These maps illustrate that each kernel was capable of detecting different features in the input image. For example, the first feature map at the top of the figure exhibits strong vertical components on the left of the character. The second feature map shows strong components in the northwest area of the top of the character and the left vertical lower area. The third feature map shows strong horizontal components in the top of the character. Similarly, each of the other three feature maps exhibits features distinct from the others.

As Figure 3 shows, the pooled maps are lower-resolution versions of the feature maps, but the former retain the basic characteristics of the latter. The volume containing these six maps is the input to the second stage. Each feature map in the second stage was generated by convolving a different kernel volume with the input volume to that stage, as explained in Figure 1. The feature maps resulting from these operations are of size 8×8; they are shown as enlarged images above the second CNN stage in Figure 3. These are not as easy to interpret visually as the feature maps in the first stage, other than to say that each exhibits a different response. Based on the accuracy of the training and test results, we know that these responses do a good job of characterizing all ten numeral classes over the entire database.

Each 192-dimensional vector resulting from vectorizing the output maps of the second stage of the CNN was fed into a fully connected net. This vector then propagated through the FCN, as explained previously. The values of the output neurons corresponding to the input image are zero or nearly zero, with the exception of the tenth neuron, whose output was 0.98. This indicates that the system correctly recognized the input image as being from the tenth class, which is the class of nines. These values of the output neurons resulted in a value for the MSE in (6) that is close to zero.

As mentioned previously in this example, training was carried out for 200 epochs. We trained the system using minibatches of 50 images between weight updates. The patterns were ordered randomly after each epoch of training, and the learning rate increment we used was $\alpha = 1.0$. This "standard" approach to training yielded excellent results in our example, but it can be refined further in more complex situations. For instance, experimental evidence suggests that large databases of RGB images containing 1,000 or more object classes require significantly deeper architectures and more complex training methodology. A good example is the deep learning neural network, *AlexNet*, that won the 2012 *ImageNet Challenge* [5].

What we have learned

After giving a brief historical account of how adaptive learning systems evolved, we introduced the basic concepts underlying the architecture and operation of deep CNNs and FCNs. The usefulness of these networks, working together to address complex image processing applications, is made possible by training the complete CNN/FCN system using backpropagation. We presented the underpinnings of backpropagation and discussed the basic equations used to implement this deep-learning scheme. The effectiveness of combining CNNs and FCNs for image pattern recognition was illustrated by training and testing a system capable of recognizing with high accuracy a large database of handwritten numeric characters.

Author

Rafael C. Gonzalez (rcg@utk.edu) received a B.S.E.E. degree (1965) from the University of Miami, FL, and M.S. (1967) and Ph.D. (1970) degrees from the University of Florida, Gainesville, all in electrical engineering. He is a distinguished service professor, emeritus in the Electrical Engineering and Computer Science Department at the University of Tennessee, Knoxville. He is a pioneer in the fields of image processing and pattern recognition and is the author or coauthor of four books, several edited books, and more than 100 publications in these fields. His books are used in more than 1,000 universities and research institutions throughout the world, and his work spans highly successful academic and industrial careers. He is a Life Fellow of the IEEE.

References

[1] F. Rosenblatt, "Two theorems of statistical separability in the perceptron," in *Proc. Symp. No. 10 Mechanisation Thought Processes,* London, 1959, vol. 1, pp. 421–456.

[2] F. Rosenblatt, *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Washington, D.C.: Spartan, 1962.

[3] D. E. Rumelhart, G. E. Hinton, R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, Vol. 1, D. E. Rumelhart et al., Eds. Cambridge, MA: MIT Press, 1986, pp. 318–362.

[4] Y. LeCun, B. Boser, J. S. Denker D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, 1989.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Advances Neural Information Processing Systems 25*, 2012, pp. 1097–1105.

[6] Y. LeCun, Y. Bengio, and G. E. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–444, May, 2015.

[7] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. New York: Pearson-Prentice Hall, 2018.

[8] E. Shelhamer, J. Long, and T. Darrell, "Fully convolutional networks for semantic segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 640–651, 2017.