CHAPTER

Deep-learning networks 15

This chapter introduces deep-learning networks. It starts with the observation that convolutional neural networks (CNNs) have gradually taken over from the more general artificial neural networks (ANNs) described in Chapter 13, Basic Classification Concepts and develops ideas on CNN architectures. With the aid of a number of key case studies, it gives reasons for the sudden explosion in the application of CNNs to classification and segmentation and also points to valuable progress being made in other areas such as automatic image caption generation.

Look out for:

- reasons why CNNs give improved performance over earlier types of ANN
- design of typical CNN architectures
- technical features such as depth, stride, zero padding, receptive field and pooling
- ways in which AlexNet was an improvement over LeNet
- how Zeiler and Fergus improved AlexNet still further
- how VGGNet achieved even better performance
- the value of deconvolution networks (DNNs) for visualizing the operation of CNNs
- the role of pooling and unpooling in CNN−DNN (encoder−decoder) nets
- how CNN−DNN nets are able to perform semantic segmentation
- how recurrent neural networks (RNNs) are used for tasks like image annotation.

One of the reasons why interest in the early ANNs waned was lack of knowledge about their internal operation and fear that they would be failure-prone because of improper or inadequate training. Eventually, such fears were dispelled by use of DNNs to help visualize the internal workings of CNNs. Coupled with this were the vastly greater datasets that have gradually become available. Today, it looks as if the future of deep networks is assured: the current problem is finding out in quite what ways they should be used vis-à-vis more conventional approaches.

15.1 INTRODUCTION

The original aim of designing ANNs was to emulate what is known to happen in the human brain, when information passes from the eyes, first to the lateral geniculate nuclei and then to the visual cortex, being processed in sequence by areas V1, V2, V4, IT (the inferior temporal cortex), and onward, until recognition is achieved and acted upon. All this appeared to happen so straightforwardly in the

Computer Vision. DOI: http://dx.doi.org/10.1016/B978-0-12-809284-2.00015-0

human brain—whole scenes being analyzed "at a glance" with no apparent effort—that it seemed worthwhile attempting it in computerized systems. Clearly, an ANN should emulate the human visual system and should consist of a number of layers, each modifying the data, first locally and then by larger and larger sets of neurons until tasks such as recognition and scene analysis are achieved. However, in the early days, full control of ANNs tended to be restricted to very few layers: indeed, a working maximum depth consisted of one input layer, three hidden layers, and one output layer—though it was later found that it should never be necessary to use more than two hidden layers, and that many basic tasks could be tackled using a three-layer network with a single hidden layer (see Section 13.12).

One of the reasons for the restriction to few layers was the credit assignment problem, which meant that it became trickier to train many layers "through" others, and at the same time, more layers meant more neurons to be trained and more computation being required to complete the task. All this being so, ANNs tended to be called on to carry out only the classical recognition process and to be fed by standard feature detectors applied in earlier nonlearning layers. Thus, the standard paradigm was that of an image preprocessor, followed by trained classifier (Fig. 15.1). As very good feature detectors could be designed by hand, this did not cause any obvious problems. However, as time went on, there was pressure to do full-scale scene analysis of real scenes, which could contain images of many types of object in many positions and poses. Thus, there was a growing need to move to much more complex multilayer recognition systems for which the early types of ANN were simply inadequate. At the same time, it became desirable to train the preprocessing system itself, so that it closely matched the requirements of the following object analysis system; clearly, it was becoming necessary to produce integrated multilayer neural networks.

Meanwhile, some workers tried other types of classifier, and ANNs were gradually ousted, support vector machines (SVMs) providing a valuable alternative: the old type of ANN was no longer up to the task and was eschewed. This happened in the late 1990s. In fact, ANNs had also had another problem, in that a "suck-it-and-see' approach was necessary when applying them to any new task: i.e., there was no scientifically based rationale for determining how many neurons or layers would be needed, how much training would be needed, or indeed quite how complex the training set would have to be. (Looking at this in another way, the ideal profile of the network was unknown: in practice profiles varied—some

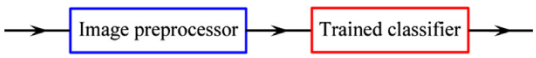


FIGURE 15.1

Classic paradigm for a classification system. The main function of the classic image preprocessor was to perform feature detection, using hand-coded, nonlearning procedures. The trained classifier would typically use ANNs trained by backpropagation.

networks being narrow and some wide in the middle.) Neither was it understood how ANNs were working internally, and few, if any methods for achieving a true scientific analysis of their operation were available. This meant that industrialists and others who might be in a position to apply them in anger did not know how reliable they would be or have the confidence to use them in real applications. In summary, traditional ANNs fell out of favor, the main reasons being the following:

1. Effective training was limited to a small number of layers and nodes per layer.
2. They had to be trained on carefully selected sets of samples, which tended to impose limits on the amount of training that could be done.
3. Their operation was not sufficiently underpinned by scientific analysis.
4. It was never certain how reliable they would be in practice, and there was significant fear that training would end up focusing on local minima.
5. SVM and other techniques had been found to outperform them.
6. In spite of the attempt to emulate biological systems, they did not scale well to direct application to large images.
7. Their architecture gave poor spatial invariance across images.

In the last case, the poor invariance arose was because in each hidden layer, the neurons were individually trained: each neuron saw different training data from the other neurons in its layer, and in any case the weights needed to be initialized randomly. This was a significant disadvantage, as it is highly preferable that the same decision should be made about any object wherever it appears in an image. The scaling problem (item 6) was also particularly serious because finding correlations between objects in a large image (e.g., finding a vehicle by locating its four wheels) required networks that scaled in size and complexity with (at least) the square of the image dimensions: it is one thing to apply an ANN to a 20×20-pixel image of a numeral in a box, and quite another to apply an ANN to a 400×400-pixel image which might well require many more hidden layers.

Overall, it was becoming clear that a different type of architecture was required to handle these direct imaging problems: in particular, it had to have spatial invariance and also it had to be able to correlate data over several scales. In principle, such a scheme would probably best be handled by starting with wide networks the same size as the input image, and looking at their outputs using smaller networks, and so on until the final layer—perhaps even a single connection—would indicate for example, that there is a fire somewhere in the viewed scene; alternatively, a small output interface could indicate where all the relevant objects are and identify them—as in the case of faces or car wheels.

In fact, there were several variants on the basic ANN idea in the late 1990s, but few of them were taken very seriously at the time and most of them died along with the other ANNs. (One should not take their "death" too literally: it merely means that they fell out of use and were effectively dead.) Nevertheless, some aficionados continued working on alternative architectures, one of which

was the CNN. This was to come to the forefront in the late 2000s, and particularly in 2011 and 2012, when the tide turned around completely, as we shall see below. The day of "deep" learning networks had finally arrived—a *deep network* being defined as one with more than three nonlinear hidden layers, which placed them beyond the scope of regular ANNs. (In an extension of this metaphor, a *very deep network* is defined as one having more than 10 hidden layers.)

15.2 CONVOLUTIONAL NEURAL NETWORKS

CNNs depart from regular ANNs in a number of key ways, which are as follows:

1. CNN neurons have local connectivity, so they do not have to be connected to *all* the outputs from the previous layer of neurons.
2. Their input fields can overlap.
3. In any layer, neurons have the same weight parameters across the whole layer.
4. CNNs abandon the old sigmoidal output function and instead use the rectified linear unit (ReLU) nonlinear function (though each convolutional layer does not have to be fed directly to a ReLU layer).
5. They intersperse convolution layers with subsampling or "pooling" layers.
6. They may have normalization layers to keep signals from each layer at suitable levels.

However, they still use supervised learning, and they still train the network by backpropagation.

Now let us consider the above differences in more detail. First, we note that a network of the general ANN type multiplies each input by an already determined weight and adds them all together. Next, if the neurons and weights are identical across a whole layer (see item 3), the resulting mathematical operation is by definition a convolution—hence, the term CNN. Regarding the overlap condition (item 2), if a given layer is to have the same dimensions as the previous layer, the input fields at each pixel will overlap almost completely (see Fig. 15.2). Interestingly, in this book, we have already seen many examples of convolutions, notably for feature detection—prominent examples being edge, corner, and interest point detection—though in each of these cases the convolutions are followed by nonlinear detectors. Item 4 deals with the necessary nonlinearity. ReLU means Rectified Linear Unit and is a nonlinear function defined by max (0, x) (see Fig. 15.3), where x is the output value of the immediately preceding convolution layer (in fact, the ReLU shadows each output connection and is a 1-to-1 filter).

Pooling (item 5) involves taking all the outputs from a locality and deriving a single output from them: usually, this takes the form of a sum or maximum operation on all the inputs. It is generally carried out in 2×2 or 3×3 windows, the former being more common, and the maximum pooling operation is more common than the sum (or averaging) operation. In each of the latter pairs of options,

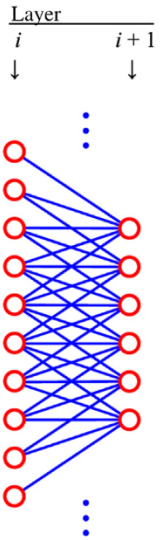


FIGURE 15.2

Part of a convolutional neural network. Note that the input fields of the neurons in layer $i+1$ overlap almost completely as they progress from layer i: here, the inputs to adjacent neurons in layer $i+1$ differ in at most 1 connection (the difference would be somewhat greater in 2-D). In this case, the input fields of the output neurons all have value 5.

FIGURE 15.3

The ReLU nonlinear function max (0, x). The function operates independently on all output connections of the immediately preceding CNN layer.

the aim is to modify the data minimally, so as to remove a significant proportion of the redundancy in a particular layer of the network, while at the same time keeping the most useful data. (Use of a maximum operation may seem somewhat surprising, as finding the maximum of a set of numbers is more likely than averaging to accentuate impulse noise: on the other hand, many convolutions contain a modicum of smoothing/low pass filtering, so the max operation need not degrade the final output significantly.)

Note that all the listed changes to the original ANN format fall within the same linear mathematical ($\Sigma_i w_i x_i$) specification, except where ReLU or pooling

operations are carried out. However, the latter still permit signal gradients to be passed through the system, so backpropagation can be applied in exactly the same way as for ANNs.

It ought to be added that several convolutional layers can be placed immediately after one another. In fact, this is equivalent to a single larger convolution. Although this arrangement may appear pointless, it can strongly affect the overall computational load. For example, three 3×3 convolutions are equivalent to a single 7×7 convolution: this means applying 27 operations instead of 49 operations, so the former implementation would appear to be more suitable. On the other hand, when the CNN calculation is implemented on a graphics processing unit (GPU—see below), this is not necessarily the case.

Overall, we can see that CNNs provide reasonable alternatives to ANNs. In addition, they seem better adapted to the model presented in Section 15.1, of moving steadily from local to global operations on images, and looking for larger and larger features or objects in the process. It is also worth emphasizing that the spatial invariance achieved by CNNs is particularly valuable and is not realizable with ANNs.

Although proceeding through the network takes us from local to more global operations, it is also common for the first few layers of a CNN to look for specific low-level features: Hence, these will typically have sizes matching that of the image. Further on in the network, it is common for pooling operations to be applied, thereby reducing the sizes of subsequent layers. After several stages of convolution and pooling, the network will have narrowed down considerably, so it is possible to make the final few layers fully connected—i.e., in any layer, each neuron is connected to *all* the outputs of the previous layer. At that stage, there are likely to be very few outputs, and those that remain will be dictated by whatever parameters must in the end be supplied by the network: these may include classifications and associated parameters such as the absolute or relative positions.

Another master-move on progressing from ANNs to CNNs is that bringing in spatial invariance greatly reduces the number of weights in the network. This makes training far more straightforward and drastically decreases the computational load for a given size of network. For a receptive field width of R, there will be only R parameters per layer, compared with a total of W in the case of an ANN. When processing 2-D images, the corresponding numbers are squared, and we have to compare R^2 with W^2 (or strictly, with WH—see the following section). Thus, we see the computational load increasing rapidly with image size for ANNs, but staying at the same low value for CNNs. In addition, the ReLU function is simpler than the ANN sigmoid function and this also speeds up processing. Indeed, it is difficult to imagine a simpler calculation than the ReLU—as we can see from the 1-line routine needed to implement it: if ($x < 0$) output = 0 else output = x. In contrast, the sigmoid function can be written in one line as output = tanh x, though this is misleading as the tanh function requires far more computation. Finally, the ReLU avoids the saturation problems that ANNs are

subject to [a neuron giving an output close to the limits (± 1) of the tanh function tends to get stuck at the same value because there is no gradient to guide the backpropagation algorithm away from it]. Indeed, the ReLU gradient is constant over the input range $x \geq 0$, and the fact that it is not attenuated over part of this range tends to speed up learning.

15.3 PARAMETERS FOR DEFINING CNN ARCHITECTURES

When analyzing CNN architectures, there are a number of points that deserve attention. In particular, several quantities and terms need to be defined—*width* W, *height* H, *depth* D, *stride* S, *zero padding* width P, and *receptive field* R. In fact, the width and height are merely the dimensions of the input image, or else the dimensions of a specific layer of the neural network. The depth D of the network or of a specific block in it is the number of layers it contains.

The width W and height H of a layer are the numbers of neurons it has in each dimension. The stride S is the distance between adjacent neurons in the output field measured in units corresponding to the distance between adjacent neurons in the input field (see Fig. 15.4): stride S can be defined along the width and

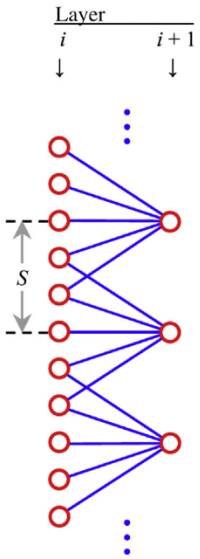


FIGURE 15.4

Illustration of the stride distance S. S is defined as the distance between adjacent neurons in the output field measured in units corresponding to the distance between adjacent neurons in the input field. Here, it is indicated by the length of the double-ended arrow. In this case, the input fields of the output neurons all have value 5, and the stride has value 3.

height dimensions but is usually the same for each. If $S = 1$, adjoining layers have the same dimensions (but see below how the size of the receptive field R can modify this). Note that increasing the stride S can be useful, as this saves memory and computation. In principle, it achieves a similar effect to pooling. However, pooling involves some averaging, while increasing the stride merely decreases the number of samples taken.

R_i is the width of the receptive field for each neuron in level i, i.e., the number of inputs for all neurons in that level. Zero padding is the addition of P "virtual" neurons providing static inputs at each end of the width dimension (here we simplify the analysis by considering only the width W of each layer, and the placement of neurons across it: ignoring the height dimension involves no loss of generality). The zero padding neurons are given fixed weights of zero, the idea being to ensure that all neurons in the same layer have equal numbers of inputs, thereby facilitating programing. However, it also ensures that successive convolutions don't lead to smaller and smaller active widths; in particular, when $S = 1$, it permits us to make the widths of adjacent layers exactly equal (i.e., $W_{i+1} = W_i$). Zero padding is actually the same concept that was used in Section 2.4 on image processing.

A simple formula connects several of these quantities: see Eq. (15.1). (It is left as an exercise for the reader to prove this formula, which is a straightforward task once the significance of each of the parameters is fully understood.)

$$W_{i+1} = (W_i + 2P_i - R_i)/S_i + 1 \tag{15.1}$$

where the suffices pertain to the inputs to layer i and the outputs feeding layer $i+1$. It is worth underlining the null situation $W_{i+1} = W_i$ that applies when $S_i = 1$, $R_i = 1$, and $P_i = 0$. (These values can be taken as base values for the design of architectures, though the next step will normally be to increase R_i to a higher value.)

By way of example, if $W_i = 7$, $P_i = 1$, $R_i = 3$, and $S_i = 1$, we get $W_{i+1} = 7$, which justifies the statement made earlier—namely that $W_{i+1} = W_i$ if $S_i = 1$. Next consider what happens when S_i is changed to 2 without altering the other parameters: in that case, we find $W_{i+1} = (7 + 2 - 3)/2 + 1 = 4$ (Fig. 15.5). Now consider what happens when W_i is changed to 9: we then find that a stride of 3 will not work (the result is not an integer) because it will not fit the width of the following layer: $W_{i+1} = (9 + 2 - 3)/3 + 1 = 3.67$. As a result, data will be lost around the edges of the image. This situation could be tackled by having different padding values (e.g., larger at the right of the image), though when R_i has a low value, the larger of the two padding values would probably not work well (the increased numbers of zeros in the corresponding receptive fields would eliminate too much information). This situation is illustrated by the last example, but now replacing $P_i = 1$ with $P_{i\mathrm{L}} = 1$ and $P_{i\mathrm{R}} = 2$ leads to a

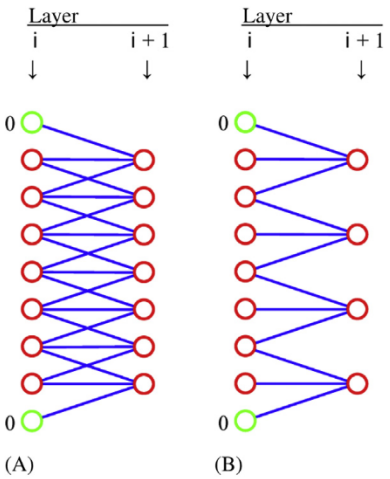


FIGURE 15.5

Details of the use of stride and zero padding for a small CNN. Part (A) shows how a seven-neuron layer is connected to the following seven-neuron layer in a case where a single "virtual" neuron (marked green) is applied at either end of the input layer to ensure that all the output neurons have the same number of input connections R. Part (B) shows what happens when the stride is increased to 2. In each case, the padding parameter $P = 1$ and $R = 3$. (See also Table 15.1, rows 2 and 3.).

width $W_{i+1} = (9 + 1 + 2 - 3)/3 + 1 = 4$. Here, we have used a slightly more general version of Eq. (15.1):

$$W_{i+1} = (W_i + P_{iL} + P_{iR} - R_i)/S_i + 1 \tag{15.2}$$

Overall, the purpose of padding is to (systematically) allow for end effects at the extremes of each layer, by ensuring that the number of zeros is adjusted to accommodate the desired stride and receptive field values. A summary of all the cases discussed above appears in Table 15.1.

Finally, an important point must be made about the definition of the depth of a number of layers of a CNN. The earlier discussion has implied that a number of adjacent layers of a CNN are normally accessed one after another in sequence—as would indeed be the case if larger and larger convolutions were implemented one after another in an effort to detect larger and larger features or even objects. However, there is another possibility—that the various layers are fed in parallel from a given starting point in the network, for example, the input image. Fig. 15.6 contrasts these two possibilities. The second possibility arises typically when an image is to be searched for a variety of different features, such as lines, edges, or corners, and the results fed in parallel to a more holistic detector. We shall see below that this strategy was adopted in the LeNet architecture, which LeCun and others developed to identify handwritten numerals and zip codes.

Table 15.1 Consistency of CNN Parameters

W_i	S_i	R_i	P_{iL}	P_{iR}	W_{i+1}	Comment
7	1	1	0	0	7	Null case
7	1	3	1	1	7	
7	2	3	1	1	4	
9	3	3	1	1	3.67	Incorrectly mapped
9	3	3	1	2	4	$P_{iL} \neq P_{Ir}$
9	3	3	0	0	3	

This table shows the values of various parameters and how they affect the number of outputs fed to the following layer of neurons. Definitions of the parameters and discussions of most of the examples below are given in the text (see also Fig. 15.5). The results are calculated using Eq. (15.2), which permits different padding values at opposite ends of the width dimension. The purpose of the formula is to permit rigorous checks to be made of the consistency of the architectural parameters: specifically; nonintegral values of W_{i+1} indicate that the architecture is incorrectly mapped.

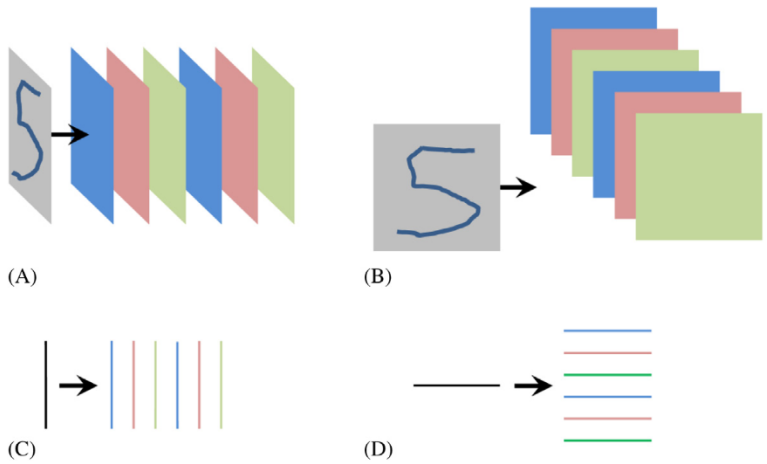


FIGURE 15.6

Visualizing depth in a CNN. Part (A) shows how an input image is applied to the first layer of a CNN: it is implied by the layout that the image is processed in turn by the six layers of the network. Part (B) shows how the image information could instead be fed in parallel to all six layers, in which case, they would act independently to locate six different sets of features in the input image. Both layouts are valid and can be represented, respectively, by the line diagrams in (C) and (D): for simplicity, the height dimension is not shown in these diagrams so that the width and depth can easily be visualized. In this example, both types of depth have the value 6.

15.4 LECUN *ET AL.*'S LENET ARCHITECTURE

In 1998, LeCun *et al.* published the now landmark case of a CNN that employed many of the stages noted above (Fig. 15.7). The input image was fed in parallel to a stack of 6 layers performing convolution operations, each immediately being followed by its own subsampling (pooling) layer; the 6 pooling layers were followed by another stack of 16 convolution layers, each followed by its own pooling layer; there then followed a sequence of two fully connected convolution layers which in turn were fully connected to a final output network containing a radial basis function (RBF) classifier.

The details of this architecture are presented in Fig. 15.7 and Table 15.2: These two forms of presentation are provided as an aid to understanding the complexities of the overall LeNet architecture, and to be sure that there is no in clarity as to what is being done. *Why* it is being done in this way is another important question which we shall also aim to understand. But first, we attend to the *what* aspect and consider some of the design features in detail:

1. The six hidden layers in LeNet are labeled C1, S2, C3, S4, C5, F6, where "C," "S," and "F" indicate respectively convolution, subsampling (pooling),

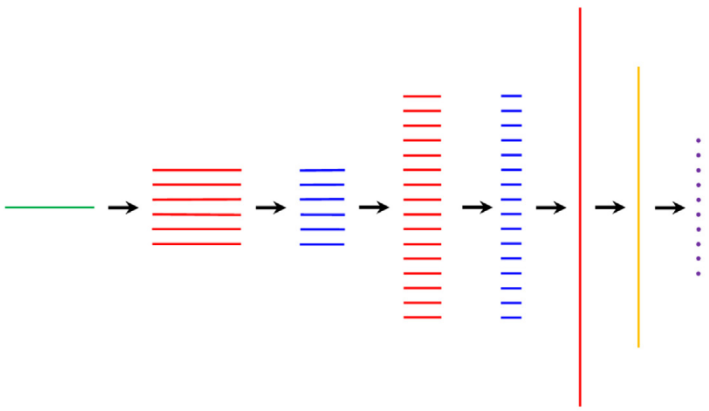


	Image	C1	S2	C3	S4	C5	F6	Output
N	1	6	6	16	16	120	84	10
$n \times n$	32×32	28×28	14×14	10×10	5×5	1×1	1	1
$r \times r$		5×5	2×2	5×5	2×2	5×5	120	84
$s \times s$		1×1	2×2	1×1	2×2	1×1		

FIGURE 15.7

Schematic of the LeNet architecture. This schematic carries much the same information as Table 15.2: both are provided as an aid to understanding the details of the overall LeNet architecture. N indicates the number of layers of each type; $n \times n$ indicates the dimensions in the case of a 2-D image format; and $r \times r$ is the size of a 2-D neuron input field (a single number indicates that the input is abstract 1-D data). Layers C5, F6, and Output are fully connected to each other.

Table 15.2 Summary of the LeNet Connections and Trainable Parameters

	Layer Size	**Connections**			**Parameters**		
Layers	***N*: $n \times n$** $r \times r$	**Inputs**	**Outputs**	**Total**	**Inputs**	**Outputs**	**Total**
Image	1: 32×32		1024 (32^2)	1024			
C1	6: 28×28 5×5	117,600 ($6 \times 28^2 \times 5^2$)	4704 ($6 \times 28^2 \times 1$)	122,304	150 (6×5^2)	6 (6×1)	156
S2	6: 14×14 2×2	4704 ($6 \times 14^2 \times 2^2$)	1176 ($6 \times 14^2 \times 1$)	5880		12 (6×2)	12
C3	16:10×10 5×5	150,000 ($60 \times 10^2 \times 5^2$)	1600 ($16 \times 10^2 \times 1$)	151,600	1500 (60×5^2)	16 (16×1)	1516
S4	16: 5×5 2×2	1600 ($16 \times 5^2 \times 2^2$)	400 ($16 \times 5^2 \times 1$)	2000		32 (60×2)	32
C5 (f.c.)	120: 1×1 5×5	48,000 ($120 \times 16 \times 5^2$)	120 (120×1)	48,120	48,000 ($120 \times 16 \times 5^2$)	120 (120×1)	48,120
F6 (f.c.)	84: 1 120	10,080 (84×120)	84 (84×1)	10,164	10,080 (84×120)	84 (84×1)	10,164
Outputs (f.c.)	10: 1 84	840 (10×84)	10 (10)	850	(Excludes RBF parameters)		–
Total (C,F,S only)		331,984	8084	340,068			60,000

LeNet has six hidden layers between the image layer and the output layer. This table gives details of the connections and the numbers of trainable parameters in each layer. Column 1 gives the name of each layer—"C," "S," and "F," indicating respectively convolution, subsampling (pooling), and fully connected convolution. The following sets of layers are fully connected (f.c.) to each other: S4–C5, C5–F6, and F6–Outputs. In each case, Column 2 indicates the number of layers N *and their dimensions:* n × n *indicates the sizes of the layers for 2-D image format, a bare "1" indicating that a layer has abstract (1-D) format;* r × r *indicates the convolution or subsampling window size. Columns 3–5 and 6–8 give the numerical values and also show, in brackets, how they are calculated. For justification of the numbers 60, 120, and 84 appearing in some of the calculations, see text. For convolutions C1, C3, and C5, the padding parameter* P = 2.

and fully connected convolution. Thus, convolution and subsampling layers alternate until C5 is reached: at that point (i.e., from layer S4 onward), the network becomes fully connected, and further subsampling layers are not needed, as *every* input connection is trained. (This explains why the numbers in the "Parameters" columns become identical to those in the "Connections" columns in Table 15.2.) S4 cannot be described as fully connected as that applies only to its outputs.

2. In Table 15.2, column 2 indicates the number of layers N of each type and their dimensions: $n \times n$ indicates that the layer has 2-D image format; a bare "1" indicates that it has abstract (1-D) format; $r \times r$ indicates the input field of each neuron. The purpose of Table 15.2 is to give details of the connections and the numbers of trainable parameters in each layer. Columns 3–5 and 6–8 give the numerical values and also show, in brackets, how they are calculated.
3. In principle, the convolution inputs and outputs should match, i.e., the numbers of inputs should be $r \times r$ times the numbers of outputs. In fact, this only happens for C1. In the other three cases, we also have to take account of the following:
   a. For C3, an attempt is made to look not just at one previous layer but at all previous layers. However, this would require inordinate amounts of computation, so a feature map is used in which the actual number of features selected is limited to 60—as indicated in detail in Table 15.3.
   b. For C5, there is a question as to why 120 layers have been used, as the LeCun et al. (1998) paper doesn't clarify this (but note that the layers are situated along the depth dimension and can nevertheless be considered as neurons). However, C5 is fully connected, and it seems likely that 120 layers constituted the largest reasonable number that could be fully connected (in the list of connections and trained parameters in Table 15.2,

Table 15.3 How the 6 Feature Maps From S2 are Fed to the 16 C4 Inputs

	1	2	3	4	5	6
1	+				+	+
2	+	+				+
3	+	+	+			
4		+	+	+		
5			+	+	+	
6				+	+	+

	7	8	9	10	11	12
1	+			+	+	+
2	+	+			+	+
3	+	+	+			+
4	+	+	+	+		
5		+	+	+	+	
6			+	+	+	+

	13	14	15
1	+		+
2	+	+	
3		+	+
4	+		+
5	+	+	
6		+	+

	16
1	+
2	+
3	+
4	+
5	+
6	+

This table indicates (by + signs) how outputs from S2 layers are selected as inputs to the C4 layers. This is achieved by combining the following selections, containing respectively 18, 24, 12, and 6 values to be carried across from S2 to C3, the total being 60. This is many more than could come from one layer of S2 but many less than the 96 that would come from full connection. The reasons for employing this sampling technique are not only to reduce memory and computation but also to include an element of randomness by breaking symmetry (too regular an arrangement might give more chance of missing important combinations of features).

120 is multiplied by 485, which leads to a total of 58,200 connections and parameters).

   c. For F6, the main consideration is making the convolution fully connected, and the number of inputs is no longer described as $r \times r$ but has the 1-D value 120. The number 84 arose because the output of layer F6 returns to image format with a 7×12 image array giving a stylized rendition of the digit in the input image. It turns out that this is useful as the usual 1-of-*N* code for classifying characters has insufficient redundancy to distinguish between dozens of possibilities: this made it better practice to convert to a stylized character format, even though the LeNet was tested primarily on digits.

4. The final output interprets the stylized 7×12 image array as one of 10 digits, using a RBF operator to make the classification.
5. The subsampling layers S2 and S4 have 2×2 inputs and perform an averaging operation. They also include a learning function with just two parameters, which respectively apply a trainable multiplying factor and a trainable bias value.
6. The 5×5 output from S4 results in C5 having a single (1×1) neuron per layer. However, it was felt preferable to take C5 as providing a 2-D output, in case the network was later increased in size. Nevertheless, given the architecture as presented in Fig. 15.7 and Table 15.2, C5 is fully connected and has been marked as such in the table.
7. "Fully-connected" means that any layer is fully connected at its inputs. (By convention, fully connected layers do not use padded inputs but merely connect all relevant inputs and outputs that contain actual signals.) We can now write down the following simple relation between the numbers of inputs and outputs:

$$\text{inputs}_{i+1} = \text{outputs}_i \times \text{outputs}_{i+1} \tag{15.3}$$

   This can be used to confirm the self-consistency of the fully connected parts of a CNN architecture and gives correct values for the numbers of input connections of C5, F6, and Output. However, when applied to C3, and ignoring the padding inputs, it gives the following input value: $inputs_{C3} = 96 \times 10^4$, which is much greater than the value given in Table 15.2 (15×10^4). This shows that C3 is far from being fully connected.
8. As a further check on the self-consistency of the architecture, we can check that the following rule applies for layer S*i*, where $i = 2$ or 4

$$\text{inputs}_i = \text{outputs}_{i-1} \tag{15.4}$$

Having dealt with the details of the architecture, it will now be useful to see why this particular architecture was chosen for the given task of numerical digit recognition.

In the handwritten Zip code and digit recognition scenario, digits appear as imprecisely located signs that are also imprecisely scaled; furthermore, they are

subject to a myriad of style variations and distortions: the latter include local malformations, incorrectly placed dots, crosses, joins, and a modicum of noise that might already be on the paper or inadvertently added after drawing, in the form of smudges, blotches, or specks of food. All these factors make it difficult for a human to read the codes correctly—and arguably no less difficult for machines. To solve these problems, it is useful to start by ensuring that reading algorithms have spatial, scale, and distortion invariance.

Fortunately, CNNs have spatial invariance built into them by the use of convolution—the same small kernel being replicated and applied right across any specific CNN layer. Similarly, many of the variations caused by scale, style, and distortion changes can be alleviated by eliminating the need for exact correspondence between the positions of small features. In particular, stacks of convolution layers each of which detects a different subfeature—such as line segments of various orientations, endpoints, crossings, and corners—can be combined in ways that are insensitive to slight changes in position. This is achieved by introducing subsampling (pooling) layers between convolutions. Thus, we can see that the LeNet architecture embodies the most important invariances by its use of stacks of convolution and pooling layers. In fact, many of the problems cited above are random in nature, and the simplest way of tackling them is by training the network on large quantities of data. However, there is one more technique that can help with this, and that is normalization of the input data by ensuring that the pooling layers also employ a trainable multiplicative coefficient and a trainable bias before passing the data through a sigmoid function. Note that if a sigmoid function is made too wide, it will become linear, tending to make the whole network linear and unable to make strong decisions; whereas if it is made too narrow, it will lose its linearity, and backpropagation training will become slow and inefficient: Clearly, a balance has to be struck between these two extremes, and it is desirable for the network to be able to make its own decisions about the best working point by learning the relevant parameters.

Next, although each pooling layer averages the inputs from the previous convolution layers and follows this by multiplying by a trainable parameter as well as adding a trainable bias, the relative weights of the inputs also have to be trained: this mechanism is required to ensure that the inputs are finely balanced for the best performance. This is achieved by including a further trainable multiplicative weight at the output of each convolution layer—as indicated in the column marked "output parameters" in Table 15.2.

In LeNet, we can now interpret layers C1−S4 as multiple stages of feature detection in image space (or in reduced scale versions of image space); in addition, we can interpret the subsequent layers as performing more abstract feature detection processes—no longer being tied to image space—and resulting in classification of the digits presented in the input images. Apart from the final RBF layer, this rather complex network was trained using the standard backpropagation algorithm: in spite of its overall depth, training did not prove intractable—as it would have been if an ANN of this size had been used. In fact, as we can now

see, the design was put together in a hand-crafted way, the various numbers of layers and their sizes being adjusted to cope optimally with Zip code and digit identification. Although the detailed weights were obtained by large amounts of training, the architecture itself was built using logic and expertise in understanding the underlying problem.

Perhaps the most interesting thing about the design is that the network is deliberately made to look for large features with *reduced* location accuracy, and if any larger network were to be devised along the same lines, this principle would probably have to be extended further.

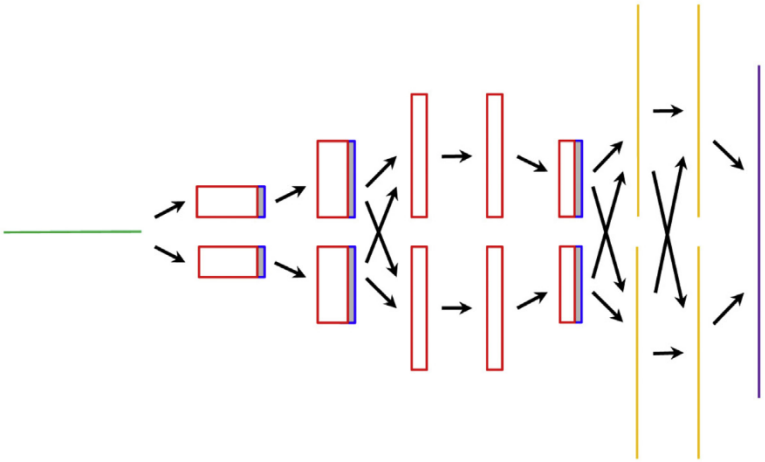
15.5 KRIZHEVSKY *ET AL.*'S ALEXNET ARCHITECTURE

AlexNet was designed specifically to target the ImageNet Challenge, which took place in 2012. (This statement is not meant to belittle the on-going projects being conducted in the designers' laboratories. However, such a challenge necessitates that, for a while, huge efforts are devoted to targeting the specific problem in hand—in this case, the ImageNet Large-Scale Visual Recognition Object Challenge (ILSVRC) in 2012.) In general, such challenges are extremely valuable as they force competitors to reach into the inner recesses of available knowledge and technology and come up with new approaches and new ideas that give the chance of making dramatic progress. Indeed, the extreme nature of these challenges leads to the possibility of making radical developments to the underlying science, and also of making breakthroughs in the technology. Nevertheless, it is best to be circumspect as one or two of the characteristics of a winning solution could still be ad hoc in nature, and therefore not worthy of slavish replication in future systems.

In this case, the AlexNet designers (Krizhevsky et al., 2012) seem to have made sound achievements at every level. First, they found it necessary to eliminate any inhibitions about using a by then quite old schema based on CNNs which hadn't obviously been at the forefront of classification research, and forcing it into shape as a winning approach. To achieve this, they had to radically improve the CNN architecture, and this necessarily gave rise to a very large software machine; they then had to speed it up dramatically with the aid of GPUs—by no means a small task as it meant reoptimizing the software to match the hardware; finally, they had to find how to feed the software system with a very large training set—again no mean task, as an unprecedentedly large number of parameters had to be trained rigorously, and several innovations were required in order to achieve this.

First, we shall attend to the architecture of the software system. We shall return later to the development of suitable training sets, and of necessary innovations for managing them and ensuring that the CNN software system is adequately trained.

The CNN architecture is shown in Fig. 15.8: Further details of the various layers are presented in Table 15.4. The number of hidden layers is 10, which is



Layer	Image	C1	C2	C3	C4	C5	F6	F7	Output
N	3	96	256	384	384	256	4096	4096	1000
n×*n*	224×224	55×55	27×27	13×13	13×13	13×13	6×6	1	1
r×*r*		11×11	5×5	3×3	3×3	3×3	1×1	1	1
s×*s*		4×4	1×1	1×1	1×1	1×1	1×1	1	1
d		3	48	256	192	192	256	4096	4096

Layer	S1	S2	S3
N	96	256	256
n×*n*	27×27	13×13	6×6
r×*r*	3×3	3×3	3×3
s×*s*	2×2	2×2	2×2
d	1	48	192

FIGURE 15.8

Schematic of the AlexNet architecture. This schematic carries much the same information as Table 15.4: both are provided as an aid to understanding the details of the overall AlexNet architecture. N indicates the number of layers of each type; $n \times n$ indicates the dimensions in the case of a 2-D image format; $r \times r$ is the size of a 2-D neuron input field (a single number indicates that the input is abstract 1-D data); $s \times s$ is the 2-D stride; d indicates the number of connections arising from the depth of the immediately preceding layer: in three cases, it is half of the actual depth (N) because the architecture is split between two GPUs. To aid clarity, subsampling layers are shaded and are shown attached to the preceding convolution layer. Layers F6, F7, and Output are fully connected.

only 4 more than for LeNet. However, these numbers are misleading as the *depths* of the various layers in AlexNet sum to 11,176 compared with 258 for LeNet. Similarly, AlexNet contains ~650,000 neurons compared with 6508 for LeNet, whereas the number of trainable parameters is some 60 million compared with 60,000 for LeNet. And when we look at the size of the input image, we find that

Table 15.4 Summary of the AlexNet Connections and Trainable Parameters

Layers	Layer Size *N*: *n* × *n* *r* × *r*: *d*	Connections: Inputs	Connections: Outputs	Parameters
Image	3: 224 × 224		150,528 (3×224^2)	–
C1	96: 55 × 55	105.42×10^6	290,400	0.03×10^6
	11 × 11: 3	($3 \times 96 \times 55^2 \times 11^2$)	($96 \times 55^2 \times 1$)	($3 \times 96 \times 11^2$)
S1	96: 27 × 27	0.63×10^6	69,984	192
	3 × 3: 48	($96 \times 27^2 \times 3^2$)	($96 \times 27^2 \times 1$)	(96×2)
C2	256: 27 × 27	223.95×10^6	186,624	0.61×10^6
	5 × 5: 48	($48 \times 256 \times 27^2 \times 5^2$)	($256 \times 27^2 \times 1$)	($96 \times 256 \times 5^2$)
S2	256: 13 × 13	0.39×10^6	43,264	512
	3 × 3: 192	($256 \times 13^2 \times 3^2$)	($256 \times 13^2 \times 1$)	(256×2)
C3	384: 13 × 13	149.52×10^6	64,896	0.89×10^6
	3 × 3: 256	($256 \times 384 \times 13^2 \times 3^2$)	($384 \times 13^2 \times 1$)	($256 \times 384 \times 3^2$)
C4	384: 13 × 13	112.14×10^6	64,896	1.33×10^6
	3 × 3: 192	($192 \times 384 \times 13^2 \times 3^2$)	($384 \times 13^2 \times 1$)	($384 \times 384 \times 3^2$)
C5	256: 13 × 13	74.76×10^6	43,264	0.89×10^6
	3 × 3: 192	($192 \times 256 \times 13^2 \times 3^2$)	($256 \times 13^2 \times 1$)	($384 \times 256 \times 3^2$)
S3	256: 6 × 6	0.08×10^6	9216	512
	3 × 3: 192	($256 \times 6^2 \times 3^2$)	($256 \times 6^2 \times 1$)	(256×2)
F6	4096: 6 × 6	37.75×10^6	4096	37.75×10^6
(f.c.)	256	($256 \times 4096 \times 6^2$)		($256 \times 4096 \times 6^2$)
F7	4096:1	16.78×10^6	4096	16.78×10^6
(f.c.)	4096	(4096×4096)		(4096×4096)
Outputs	1000:1	4.10×10^6	1000	–
(f.c.)	4096	(1000×4096)		(softmax)
Total (C,F only)		720.32×10^6	658,272	58.28×10^6

AlexNet has 10 hidden layers between the image layer and the output layer. This table gives details of the connections and the numbers of trainable parameters in each layer. Column 1 gives the name of each layer—"C," "S," and "F," indicating respectively convolution, subsampling (pooling), and fully connected convolution. In each case, Column 2 indicates the number of layers N *and their dimensions:* n × n *indicates the sizes of the layers for 2-D image format, a bare "1" indicating that a layer has abstract (1-D) format;* r × r *indicates the convolution or subsampling window size. C1 has a 4 × 4 stride; the three subsampling windows have a 2 × 2 stride. Columns 3–5 give the numerical values and also show, in brackets, how they are calculated. Bias parameters are presumed to number 1 per neuron output for convolution layers (C, F) and 2 per output for subsampling layers (S), as for LeNet.*

AlexNet takes a color image of size 224 × 224, whereas LeNet could only manage a bilevel 32 × 32 input image. So overall, AlexNet is larger than LeNet by a factor between 100 and 1000, depending on which factors should be regarded as the most relevant. However, the real change wrought by AlexNet was the

possibility of working with huge numbers of layers and managing the credit assignment problem in spite of this—while still using the backpropagation algorithm for training. At the time, this was unprecedented, but it was made possible partly by the already reduced number of parameters required by CNNs, because, as we have seen, all the neurons in any given layer of neurons are forced to use identical parameters; it was also made possible by employing exceptionally large training sets, and by other methods to be described below.

Before proceeding further, we shall examine various details of the AlexNet architecture. One prominent feature of the architecture is the horizontal split right across the network, above and below which a single GPU is used for implementation (Fig. 15.8). In principle, this should impose serious limitations upon the architecture, but in practice, it turned out not to be an insoluble problem. (In fact, this could perhaps be regarded as one of the most cunning innovations in the system.) This is because it is reasonable to let the each GPU chunter away at half the image and its features, and because at one juncture (between layers C2 and C3) data from the other half of the data is brought back again; as an added measure, all the data is brought together once again for the final two fully connected layers (F6 and F7)—and for the fully connected Output layer embodying the softmax computations. Next, note that all the convolution layers take in the neuron $r \times r$ fields of view for *all* the available depth outputs from the previous layer. However, in layers C2, C4, and C5, the *available* depth outputs d number only half of those from the previous layers—because transference of data across the inter-GPU divide is not possible: i.e., in those three cases, d is only half the value of N for the previous layer.

There is also a definite need to reduce redundancy in the system at an early stage. This is achieved rather brutally by applying a 4×4 stride in layer C1, though excessive damage is prevented by applying an 11×11 pixel window to help gather sufficient information at the same time. In fact, this is the only place in the whole system where a stride of greater than 1 is used—except in the case of the three subsampling layers S1, S2, S3. Interestingly, excessive damage is prevented in the latter cases by using a 3×3 rather than the more usual 2×2 pooling window. These three subsampling layers are termed "overlapping pooling," and are particularly easy to visualize (Fig. 15.9): in fact, they employ max-pooling rather than averaging.

Notice that the dimensions $(n \times n)$ of the layers fall rapidly first from 224×224 to 55×55, then successively to 27×27, 13×13 (three times), 6×6 and finally down to 1×1. The published paper (Krizhevsky et al., 2012) makes no mention of padding niceties and perhaps this was due to the need to the rush to complete the machine in 2012 (of course, the published paper had to report what was actually done, rather than to overidealize it). But in any case rapid convergence to size 1×1 and thereby to purely abstract pattern classification processes is of value, both for minimizing storage and for maximizing speed. Interestingly, almost all the trainable parameters are in layers F6 and F7, only 1000 inputs being left for the final softmax (nonneural) classifier.

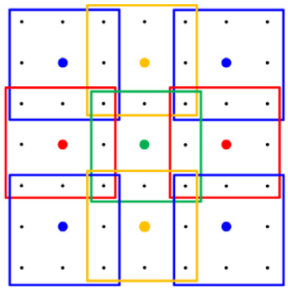


FIGURE 15.9

A simple case of overlapping pooling. This figure shows a small 7×7 pixel image (small dots) in which the sampled output points (large dots) represent a 2×2 stride mapping. The squares show how each output point is fed from a 3×3 pooling window. To prevent confusion, the square windows are offset slightly from each other, but they all still contain nine pixel centers.

One of the features of AlexNet that set it apart from LeNet was the use of the then recently developed ReLU nonlinear transfer function. Krizhevsky et al. found that this was able to speed up training by a factor of about 6 relative to the usual *tanh* function. As huge amounts of training were needed in AlexNet, this was a valuable innovation. In fact, ReLUs do not need input normalization to prevent them from saturating (it is obvious that a linear response can never saturate). Nevertheless, Krizhevsky et al. found that it was still useful to include an element of what they called "brightness normalization," which they considered to perform a function similar to that of lateral inhibition in the human visual system. They found that including it improved error rates by between 1% and 2%.

Very shortly before AlexNet was completed, a new technique called "dropout" was introduced by Hinton et al. (2012): see also Hinton (2002). The purpose of this was to limit the incidence of overtraining. This was achieved by randomly setting a proportion (typically as high as 50%) of the weights to zero for each training pattern; this rather surprising technique appeared to work well: it did so by preventing hidden layers from relying too much on the specific data fed to them. (Another way of looking at this is that random sampling from 2^{WH} different architectures is taking place, so the possibility of any of them being overtrained should be negligible: in effect, the network is being trained via many independent pathways, so any individual overtrained pathway should not affect the overall performance.)

Krizhevsky et al. (2012) included this feature in AlexNet. To apply it, the output of each neuron is randomly set to zero with probability 0.5. This is done before the forward pass of the input data, and the affected neurons do not contribute to the ensuing backpropagation. On the next forward pass, a different set of neuron outputs is set to zero with probability 0.5, and again the affected neurons do not

contribute to backpropagation; and similarly for later passes. During testing, an alternate procedure occurs, with all neuron outputs being multiplied by 0.5. In fact, multiplying *all* neuron outputs by 0.5 is an approximation to taking the statistical geometric mean of all the local neuron output probability distributions and relies on the geometric mean being not too far from the arithmetic mean. Dropout was incorporated into the first two layers of AlexNet, and significantly reduced the amount of overfitting (undertraining because of too little training data): the main disadvantage of including it was to double the number of iterations needed for convergence. Effectively, the training time was doubled but the effectiveness of training was significantly improved—all of which was far better than the usual outcome of excessive training resulting in poorer performance!

AlexNet was trained using the 1.2 million images available from the ImageNet ILSVRC challenge, this number being a subset of the full 15 million in the ImageNet database. In fact, ILSVRC-2010 was the only subset for which test labels were available, there being about 1000 images in each of 1000 categories. However, it was found that these images were far too few to specify a CNN of the complexity required to perform accurate classification of this immense task. Therefore, means were required for expanding the dataset sufficiently to train AlexNet and achieve classification error rates in the 10% to 20% bracket.

Two main means of augmenting the dataset were considered and implemented. One was to apply realistic translations and reflections to the images in order to generate more images of the same type. The transformations even extended to extracting five 224×224 patches and their horizontal reflections from the initial 256×256 ImageNet images, giving a total of 10 patches per image. Another was to alter the intensities and colors of the input images. To make this exercise more rigorous, it was carried out by first using principal components analysis (PCA) to identify color principal components for the ImageNet dataset, and then to generate random magnitudes by which to multiply the eigenvalues, thereby producing viable variants of the original images. Together, these two approaches were able to validly generalize and multiply the size of the original dataset by a factor of ~ 2000—the principle being to generate natural changes in position, intensity, and color (the latter would naturally change with the color of the ambient illumination).

At this stage, it ought to be emphasized that the aim of the challenge was to find the best vision machine (giving the lowest classification error rates) that is able to recognize an example of a flea, a dog, a car, or other common object in any position in an image and in any reasonable (i.e., naturally occurring) pose. Furthermore, the machine should prioritize its classifications to give at least the first five most probable interpretations. Then, each machine can be rated not only on the accuracy of its top classification but also on whether the object classified appears within the machine's top five classifications. AlexNet was able to achieve a winning top-5 error-rating of 15.3%, compared with 26.2% for the runner up. Another first was the dramatic drop to below 20% error rate for such an exercise, which spelt a new lease of life for deep neural networks, and brought them sharply back into the limelight.

Note that all this was achieved not merely by designing a winning architecture and generating the right dataset to train it adequately, but also it was necessary to bring the training time down to realizable levels. In this respect, GPU implementation proved to be crucial. Even with a pair of GPUs, the training time took approximately a week, working 24 hours a day, to manage the task. Without GPUs, it would have taken some 50 times longer—most probably, about a year—so the machine would have had to be submitted to the following year's challenge! [A commonly accepted figure is that a GPU has a speed advantage ~ 50 relative to a typical host CPU.]

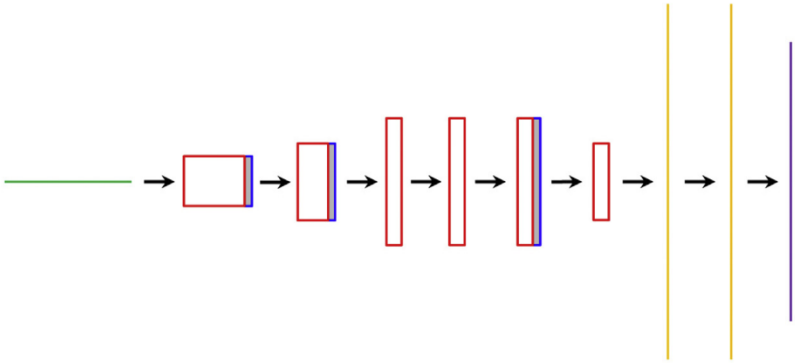
Finally, it should be noted that GPUs provide a very good implementation because of their intrinsic parallelism, and thus their capability for handling large datasets in fewer cycles. Note that each layer of a CNN is completely homogeneous and is therefore well adapted for parallel processing. Note also that GPUs are well adapted to working in parallel as they are able to read from and write to each other's memories directly, avoiding the need to move data through the host CPU memory.

15.6 ZEILER AND FERGUS'S WORK ON CNN ARCHITECTURES

Following the almost unprecedented success of AlexNet relative to previous CNN architectures such as LeNet, many workers concentrated on consolidating this progress and augmenting it further. In particular, Zeiler and Fergus (2014) made a detailed analysis of its optimality and means for improving it. They started with statements including the following, "There is no clear understanding of why [CNNs] perform so well, or how they might be improved;" "there is still little insight into the internal operation and behavior of these complex models, or how they achieve such good performance;" "from a scientific standpoint, this is deeply unsatisfactory."

Their most immediate task was to reimplement AlexNet and to test it thoroughly, with the aim of finding its limitations and developing it further. They employed the same ImageNet 2012 training data and augmented it in the same way—by resizing the images, cropping to 256×256, and using 10 different subcrops to 224×224 pixels, again incorporating horizontal flipping. The main difference between the new implementation and AlexNet was the use of a single GPU in place of the original two. This allowed Zeiler and Fergus to replace the sparse connections between layers C3 and C5 with a complete set of dense connections. In addition, they used a powerful visualization technique (described in more detail in the next section) to improve performance further. One such improvement was to renormalize the various filters in the convolutional layers to prevent any of them dominating (effectively, a dominant layer prevents other layers from contributing fully to object recognition).

Visualization also helped with the selection of optimal architectures. In particular, it showed that the first and second layers of AlexNet were weak in the middle spatial-frequency range—with hindsight, this was a fairly obvious result of jumping straight from an 11×11 filter of large (4×4) stride to a 5×5 filter with a 1×1 stride: they also showed that the large stride was the cause of aliasing problems, which had not been anticipated. They dealt with these problems by placing a 7×7 filter in C1 and reducing the stride to 2×2. However, these changes necessitated modifications in later layers, because the image size had not been reduced sufficiently in the first few layers: this will be seen from the ZFNet architecture schematic shown in Fig. 15.10; we name this the "ZFNet architecture" to avoid confusion with AlexNet and other architectures. As a result of all



Layer	Image	C1	C2	C3	C4	C5	C6	F7	F8	Output
N	3	96	256	384	384	256	256	4096	4096	1000
$n \times n$	224×224	110×110	26×26	13×13	13×13	13×13	6×6	6×6	1	1
$r \times r$		7×7	5×5	3×3	3×3	3×3	3×3	1×1	1	1
$s \times s$		2×2	2×2	1×1	1×1	1×1	1×1	1×1	1	1
d		3	96	256	384	384	256	256	4096	4096

Layer	S1	S2	S3
N	96	256	256
$n \times n$	55×55	13×13	6×6
$r \times r$	3×3	3×3	3×3
$s \times s$	2×2	2×2	2×2
d	1	96	256

FIGURE 15.10

Schematic of the ZFNet architecture. This schematic is very similar to that for AlexNet, as shown in Fig. 15.8. Indeed, it is so close that the meanings of all terms should be clear from Fig. 15.8. Notice that AlexNet contains seven hidden layers, whereas ZFNet contains eight hidden layers (these figures are exclusive of the three subsampling layers in each network). Note that ZFNet is implemented using only a single GPU, and its architecture is not split. Hence, d is now equal to the number of connections arising from the depth of the immediately preceding layer. The main feature to focus on here is how the values of $n \times n$, $r \times r$ and $s \times s$ vary between AlexNet and ZFNet.

these changes, ZFNet was able to achieve a top-5 error-rating of 14.8%, which should be compared with the AlexNet figure of 15.3%. (It turned out that Clarifai did even better in the 2013 ImageNet competition, attaining a figure of 11.7%, but space precludes including a detailed discussion of that story here.)

Zeiler and Fergus (2014) carried out further experiments to explore the reasons for the success of the AlexNet architecture. They tried adjusting the sizes of the different layers and also removing them entirely—each time completely retraining the architecture on the same data. Many of these changes made little difference to the overall performance, leading only to slight increases in the error rate. Nevertheless, removing several layers at once led to much worse performance, whereas increasing the size of the middle convolution layers improved performance significantly. They concluded that maintaining the overall depth of the architectural model, whereas making detailed changes is important for achieving good performance. One can argue that the "intelligence" of a net intrinsically increases with depth, but to capitalize on this intelligence requires increased training and thus substantially increased computation.

This sort of approach is sometimes called an "ablation study," the word "ablation" being a commonly used medical term meaning, shaving off diseased flesh, layer by layer. Applied to classification systems such as CNNs, progressive removal of layers aims to reveal which layers are critical to the architecture. Interestingly, in this case, it revealed something different—that the minimum depth is more critical rather than any individual layer.

To proceed further, Zeiler and Fergus showed that it was possible to use the previously trained CNN layers to perform totally different tasks using the Caltech-101, Caltech-256, and Pascal VOC 2012 datasets, whose main characteristics we now briefly describe:

- ***Caltech-101***: This dataset contains pictures of objects in 101 categories, with $\sim$50 images per category, though some categories have up to $\sim$800 images. The image size is $\sim 300 \times 200$ pixels. This dataset was collected in 2003 by Fei-Fei Li et al. The images have little clutter and are approximately centered.
- ***Caltech-256***: This dataset contains pictures of objects in 256 categories and contains a total of 30607 images, i.e., around 120 per category, with a minimum per category of between 30 and 80. The dataset has two valuable features: artefacts due to image rotation are avoided and a substantial clutter category is introduced for testing background rejection. The dataset was published in 2006 by Griffin et al.
- ***PASCAL VOC 2012***: This dataset contains 20 classes. Note that the images can contain several objects: the training/validation data has 11,530 images containing 27,450 ROI annotated objects and 5,034 segmentations. However, the annotations are incomplete: only people are annotated and some people are not annotated. See van de Sande et al. (2012) for further details.

To achieve this, they merely retrained the softmax output classifier appropriately: as this classifier contains relatively few parameters, it can be trained quite

quickly. The retrained ZFNet system performed better on the two Caltech datasets than the best previously reported performance, whereas on the PASCAL dataset, it performed much less well. This was because the PASCAL dataset images can contain multiple objects, whereas the ZFNet system only provided a single exclusive prediction for each image. Concentrating on the two Caltech datasets, the considerable success achieved was presumably due to the extensive training of the CNN core: this couldn't be matched by the previous leading systems because of the paucity of the data they had available for training. On the other hand, when ZFNet was retrained on this data, it performed poorly in each case: this reflects the enormous thirst for training data of such a powerful neural classifier.

15.7 ZEILER AND FERGUS'S VISUALIZATION EXPERIMENTS

We now move on to Zeiler and Fergus's visualization experiments. These were designed to reveal the inner workings of CNNs. Basically, they sought to analyze quite how a properly trained CNN processes incoming data. It proved difficult to do this holistically; in fact, it could only be done one layer at a time. What they needed to do was to feed a new image into the system, look at the output from layer i, and try to work out what happened to the signals in passing through that layer (Fig. 15.11). Suppose that at that point the data has just passed through a

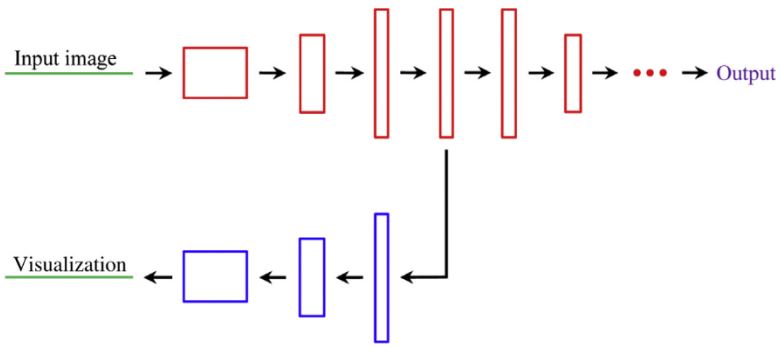


FIGURE 15.11

The idea of a deconvolution network. This schematic shows (top) the normal pathway from the input image to the output. This pathway is trained as usual by backpropagation, and the learnt parameters are then fixed. Next, a new input image is applied, and data is extracted laterally and passed to the visualization pathway. The aim is to deconstruct the learnt parameters to see how the new input image activates the network. However, the latter process is more complicated than might initially be imagined, as pooled maxima have to be unpooled (i.e., passed back to the correct channels), reversed ReLUs have to be included, and convolution coefficients have to be "deconvolved."

subsampling layer, and specifically its max-pooling layer. At that point, it is unknown which of the $r \times r$ inputs (typically, a total of 4 or 9 inputs) gave rise to the maximum signal. However, for that particular input image, we need to go down to the next layer to determine which it is. Then, we need to go one stage further, to the next lower layer, and track the maximum signal even further. At each layer, the unit that does the backward recursion (in the original Latin sense of running back, not of using a recursive procedure that calls itself) is called an "unpooling" layer, and as it operates it invokes so-called "switches" that locate the path back to the previous maximum.

Looking more closely at the process of downwards recursion, the whole downwards orientated system is actually put into reverse—not merely the pooling operation. Even the ReLUs are pointed downwards—as they have to ensure that the signals moving downwards remain mutually compatible and positive. (Although some aspects of the inversion process may seem strange and somewhat arbitrary, much will become clear from the revised approach adopted in the following section.) Lastly, the convolution filters themselves have to be inverted by using "transposed" versions of the filters (in practice, this is approximately achieved by rotating them through 180°: Zeiler and Fergus, 2014). Interpreting the resulting pictures can be quite difficult. However, the observation that signals in AlexNet's middle spatial-frequency range were weak is typical of what is possible with this approach. Similarly, occlusion sensitivity can be analyzed to find quite where in the trained network crucial losses in activity occur as a result of (partial) occlusion.

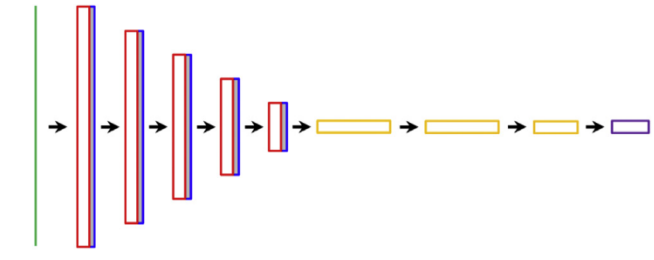
Although the work carried out on visualization is highly useful to answer questions such as these about the operation of the CNN, it is also rather limited. In particular, it applies only to the particular image input at any one time: i.e., it only visualizes a single activation and does not automatically carry lessons about the overall operation of the network. As a result, only an approximate version of the convnet features from the layer underneath the current location in the network can be recovered from the system.

Finally, note that the complete system architecture that carries out the visualization is called a "DNN:" this is because it includes not only the components of the original convolutional network but also, at the same time, the same components used in reverse. Thus, it contains unpooling elements as well as pooling elements and also has to reverse both the ReLUs and the convolutions. In the case of convolutions, it is impossible to achieve exact reversal, for the same reason that removing blur from images cannot be done exactly: however, see Zeiler et al. (2010) for a way forward using a regularization approach. Overall, what Zeiler and Fergus have achieved is (1) to significantly improve our confidence about the operation and validity of CNNs and (2) to identify a number of viable methods for improving their performance. Nevertheless, advance design of all the details of a CNN in terms of the number of layers, and profiles of the image sizes ($n \times n$), and the neuron input field sizes ($r \times r$)—not to mention optimum stride sizes ($s \times s$)—still seems rather out of reach.

15.8 SIMONYAN AND ZISSERMAN'S VGGNET ARCHITECTURE

In the continued absence of knowledge about the form of an ideal architecture, Simonyan and Zisserman (2015) set out to determine the effect of further increases in depth. To achieve this, they significantly reduced the number of parameters in the basic network by limiting the maximum neuron input field to 3×3. In fact, they restricted the convolution input field and stride to 3×3 and 1×1, respectively, and set both the input field and the stride of each subsampling layer to 2×2. In addition, they arranged for the systematic and rapid convergence of the successive layers from 224×224 down to 7×7 in 5 stages, followed by a transition to 1×1 in a single fully connected stage; this was followed two further fully connected layers and then a final softmax output layer (Fig. 15.12). All the hidden layers included a ReLU nonlinearity stage (not shown in the figure): local response normalization—which had been used by Krizhevsky et al. in AlexNet—was excluded as it was found not to improve performance. Apart from the N "channels," the 5 convolutional layers C1−C5, respectively, contained 2, 2, 3, 3, 3 identical sublayers, though these are not marked in Fig. 15.12. Finally, it should be remarked that, for reasons of experimentation, Simonyan and Zisserman devised six variations on the VGGNet architecture, with 11 to 19 weighted hidden layers: Here, we only cover configuration D (with 16 weighted hidden layers), for which the numbers of identical sublayers in layers C1−C5 are as listed above. Configuration D is of particular interest as it was used in the work of Noh et al. (2015), to be described in detail in the next section.

As mentioned above, Simonyan and Zisserman saved on the basic number of parameters by restricting the convolution input field to 3×3. This meant that larger convolutions had to be produced by applying several 3×3 convolutions in sequence. Clearly, a 5×5 input field would necessitate applying two 3×3 convolutions, and a 7×7 field would require three 3×3 convolutions. In the latter case, this would reduce the total number of parameters from $7^2 = 49$ to three times $3^2 = 27$. In fact, not only did this way of implementing a 7×7 convolution reduce the number of parameters, but also it forced an additional regularization on the convolution, as a ReLU nonlinearity was interposed between each of the 3×3 component convolutions. It is also relevant that both the input and output of each three-layer 3×3 convolution stack can have N channels, in which case it will contain a total of $27N^2$ parameters, and it is that figure which should be compared with $49N^2$ parameters. The numbers of parameters in the various layers of VGGNet (configuration D) are given in Table 15.5, the underlying formula for C1−C5 being $M \times 3^2N^2$ for a stack of M 3×3 convolutions. However, note that this formula only applies if the previous convolution layer has the same value of N, which is valid for C5 but is not the case for C1−C4: see Table 15.5 for the detailed picture.



Layer	Image	C1	C2	C3	C4	C5	F6	F7	F8	Output
N	3	64	128	256	512	512	4096	4096	1000	1000
$n \times n$	224×224	224×224	112×112	56×56	28×28	14×14	7×7	1	1	1
$r \times r$		3×3	3×3	3×3	3×3	3×3	1×1	1	1	1
$s \times s$		1×1	1×1	1×1	1×1	1×1	1×1	1	1	1

Layer	S1	S2	S3	S4	S5
N	64	128	256	512	512
$n \times n$	112×112	56×56	28×28	14×14	7×7
$r \times r$	2×2	2×2	2×2	2×2	2×2
$s \times s$	2×2	2×2	2×2	2×2	2×2

FIGURE 15.12

Architecture of VGGNet. This architecture shows a more recent optimization of the standard type of CNN network. Unlike the schematics in Figs. 15.7, 15.8, and 15.10, this one is arranged to show the relative sizes of the convolution layers, which range from image size down to 1×1. Note that the convolution layers all have unit stride, and that their input fields are limited to a maximum size of 3×3: the subsampling layers all have 2×2 input fields and 2×2 strides.

VGGNet has a number of possible configurations, with numbers of weighted hidden layers ranging from 11 to 19. Configuration D (shown above) has 16 layers, layers C1—F8, respectively, containing 2, 2, 3, 3, 3; 1, 1, 1 weighted sublayers. The number of weighted layers determines the number of parameters: see Table 15.5.

In spite of its increased depth, VGGNet contains only about 2.4 times as many parameters as AlexNet, it is much simpler and doesn't split the architecture to fit it to a 2-GPU system (see Fig. 15.8). On the contrary, it immediately obtains a speedup of 3.75 times over a single GPU when using an off-the-shelf 4-GPU system.

Details of the training methodology are similar to those for AlexNet: see the original paper by Simonyan and Zisserman (2015). However, these authors include one interesting innovation: that is to use "scale jittering," while training—i.e., to augment the training set using objects over a wide range of scales. In fact, random scaling was applied over an image scale factor of 2.

The outcome was that VGGNet achieved top-5 test error results of 7.0% using a single net, compared with 7.9% for GoogLeNet (Szegedy et al., 2014). In fact, GoogLeNet achieved a figure of 6.7%, but only by employing seven nets. Thus,

Table 15.5 VGGNet Parameters

Configuration D	*N*	Sublayer	Formula	Parameters
C1	64	1	$(3 \times 3) \times 3 \times 64$	0.04×10^6
		2	$(3 \times 3) \times 64^2$	
C2	128	1	$(3 \times 3) \times 64 \times 128$	0.22×10^6
		2	$(3 \times 3) \times 128^2$	
C3	256	1	$(3 \times 3) \times 128 \times 256$	1.47×10^6
		2	$(3 \times 3) \times 256^2$	
		3	$(3 \times 3) \times 256^2$	
C4	512	1	$(3 \times 3) \times 256 \times 512$	5.90×10^6
		2	$(3 \times 3) \times 512^2$	
		3	$(3 \times 3) \times 512^2$	
C5	512	1	$(3 \times 3) \times 512^2$	7.08×10^6
		2	$(3 \times 3) \times 512^2$	
		3	$(3 \times 3) \times 512^2$	
F6	4096		$(7 \times 7) \times 512 \times 4096$	102.76×10^6
F7	4096		4096×4096	16.78×10^6
F8	1000		4096×1000	4.10×10^6
Total				$\mathbf{138.35 \times 10^6}$

This table summarizes the numbers of parameters in the various convolution layers of VGGNet, configuration D. Note that most of the parameters appear in the early fully connected layers, particularly F6.

VGGNet achieved second place in the ILSVRC-2014 challenge. However, after submission, the authors managed to decrease the error rate to 6.8% using an ensemble of two models—substantially the same performance as for GoogLeNet, but with significantly fewer nets. Interestingly, all this was achieved even though the VGGNet architecture did not depart from the classical LeNet architecture of LeCun et al. (1989), the main improvement being the significantly increased depth of the network.

In spite of being placed second in the ILSVRC-2014 challenge, VGGNet has proved more versatile and adaptable to different datasets and is a preferred choice in the vision community for extracting features from images. This seems to be because VGGNet actually provides more robust features even though it turned out to have slightly weaker classification performance on a specific dataset. As we shall see in the next section, VGGNet was the network chosen by Noh et al. (2015) for their work on DNNs.

15.9 NOH *ET AL.*'S DECONVNET ARCHITECTURE

Inspired by the work of Zeiler and Fergus, as outlined in Sections 15.6 and 15.7, Noh et al. (2015) produced a "learning DNN" (DeconvNet) that learnt *from*

training how to deconvolve the sets of convolution coefficients in each layer of a CNN. In fact, the overall architecture of their system seems inspired and far simpler than indicated by the explanations in Section 15.7: Once again, hindsight radically clarifies the progress made. Their architecture is shown in Fig. 15.13: note that its initial CNN section is borrowed from layers C1–F7 of VGGNet but excludes layer F8 and the Output softmax layer. In this architecture, we see no downflowing paths, but that is because the partial downflowing path of Zeiler and Fergus's deconvolution (deconv) network has been rotated, extended, and added after the upflowing section. Indeed, it can also be said that this architecture is a

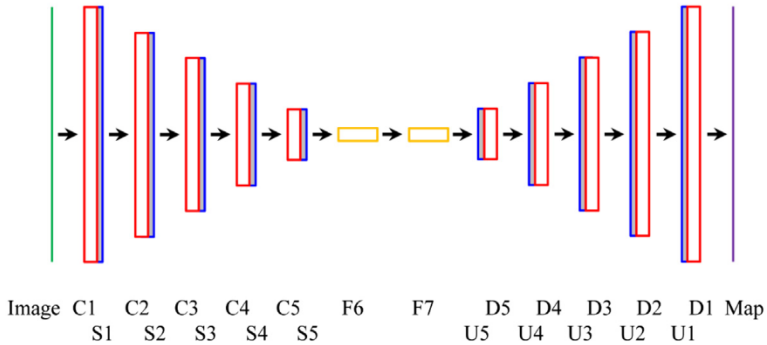


FIGURE 15.13

Schematic of Noh et al.'s learning deconvolution network. This network contains two networks back to back. On the left is a standard CNN network, and on the right is the corresponding DNN "deconvolution" network that seems to operate in reverse. The CNN network (on the left) has no output (e.g., softmax) classifier, as the ultimate purpose is not to classify objects but to present a map of where they are on a pixel-by-pixel basis throughout the area of the image. Deconv layers D5 down to D1 are intended to progressively unpick layers C5–C1. Similarly, unpooling layers U5 down to U1 are intended to progressively unpick pooling layers S5–S1. To achieve this, the position parameters from the max-pooling layers have to be fed to corresponding locations in corresponding unpooling layers (i.e., locations from S*i* should be fed to U*i*).
This schematic is arranged to show the relative sizes of the convolution layers, which range from image size down to 1×1, and then up again to give a full-size segmentation map of the input image. Overall, this architecture can be regarded as a single CNN and could in principle be trained accordingly. However, its large depth dictates that a different strategy needs to be adopted (see text for further details).
Note that the CNN network was borrowed from VGGNet (Fig. 15.12), but excluded both its layer F8 and its Output softmax layer. For details of layers C1–F7 and S1–S5, see Fig. 15.12. The details of D*i* are as for C*i*, and those for U*i* are as for S*i*—except that D*i* and U*i* expand rather than contract the dataflow. In many ways, it is useful to regard the expansion process as *mirroring* the original contraction process. Although suitable combination rules are needed to define what happens with overlapping output windows, the system is *trained* to perform the relevant deconvolutions.

generalization of Zeiler and Fergus's deconv network. It will be useful to provide a rationale for this. First, an upflowing CNN is needed for identifying objects in the input image. Second, if objects are to be located in particular parts of an image, another CNN is needed to point to the positions, and this necessarily has to follow the identification process (a network aimed at achieving all this is called a semantic segmentation network). Undertaking both tasks in one enormous unconstrained CNN would be prohibitive of memory and training, so the two networks have to be linked closely together. The means of connecting them together is to provide feedforward paths from the pooling units to later unpooling units. So the very means by which the CNN output was generalized to *eliminate* the effects of sample variations has led to the second CNN being augmented to *yield* the required location maps. Crucially, we also see that the overall single upflowing data-path makes it obvious why the ReLU units must now all point in the same direction. (Remember that they have now all been turned to face forwards again.) It is also clear that with such a huge network, training will have to be carried out carefully, and it would appear obvious that the originally upflowing (object detection) section should initially be trained on its own.

Overall, the system works by mirroring the input CNN by including a DNN after it. The operation may be summarized as follows: unpooling layer U*i* is nonlinear and redirects (unpools) the C*i* maximum signals; then deconvolution layer D*i* operates linearly on the data, and therefore has to sum the overlapping inputs, weighted as necessary. However, rather than constructing suitable combination rules to define what happens with the overlapping output windows of each D*i* layer—and doing this in some very approximate way (such as "transposed" versions of the convolution filters)—the deconvolution layers are trained as normal parts of the overall network. Although this is a rigorous approach, it does increase the burden involved in training the network.

It is also useful to have a mental model of the whole process occurring in the DNN. First, each unpooling layer recovers the information from the corresponding pooling layer and reconstitutes the dimensions the dataspace had before pooling. However, it only populates it sparsely, with the local maximum values in appropriate positions. The purpose of the following deconvolutional layer is to reconstruct a dense map in its dataspace. So, although the CNN reduces the size of the activations, the following DNN enlarges the activations and makes them dense again. However, the situation is not completely unraveled, as only the maximum values have been reinserted. As Noh et al. (2015) say in their paper, "unpooling captures *example-specific* structures by tracing the original locations with strong activations back to image space," whereas "learned filters in the deconvolutional layers tend to capture *class-specific* shapes." What this means is that the deconvolutional layers rebuild the example shapes to correspond more accurately to what would be expected for objects of the specific classes.

In spite of this assurance, the network has to be trained appropriately. However, the surmise given above about training in two stages has been improved by Noh et al. as follows: to beat the problem of the space of semantic

segmentation being extremely large, the network is first trained on easy examples, and then trained on more challenging examples: this amounts to a sort of bootstrapping approach. In more detail, the first of the training processes involves limiting the variations in object size and location by centering and cropping them in their bounding boxes; the second stage involves ensuring that the more challenging objects are adequately overlapped with ground-truth segmentations: to achieve this the widely used intersection over union measure is used and is taken to be acceptable only if it is at least 0.5.

In fact, the first stage uses a "tight" bounding box, and this is extended by a factor 1.2 and further expanded into a square in order to include sufficient local context around each object. In this first stage, the box is rated according to the object located at its center, other pixels being labeled as background. However, in the second stage, this simplification is not applied, and all relevant class labels are used for annotation.

The method is shown to be more accurate than the earlier "fully convolutional" network (FCN) of Long et al. (2015): these have respective mean accuracies of 70.5% and 62.2%. (Space precludes a full explanation of the FCN here. However, it may be envisaged as a simplified network constructed of convolution layers, including stride and pooling, but with no unpooling, no final pooling layer, and no final classifier layer—all fully connected layers having been converted to convolutions.) However, Noh et al. also show that the two types of network are, to a significant degree, complementary, and combining them into an ensemble gives better results (72.5%) than either taken alone. Specifically, FCN is better at extracting the overall shape of an object, whereas DeconvNet is better at capturing the fine details of the shape. To achieve the best results, the ensemble method takes the mean of the output maps from the two methods and then applies a conditional random field (CRF) to obtain the final segmentation. (Taking the mean of the output maps is justified, as the output maps of FCN and DeconvNet are class-conditional probability maps, each computed independently from the input image. Note also that in computer vision, CRFs are often used for object recognition and image segmentation: a CRF can take context into account by the use of prior probabilities.) In view of the slightly confused situation, we shall not pursue either of these methods further here: instead, we look at another closely related method—that of Badrinarayanan et al. (2015), which uses much less memory and has several other advantages.

15.10 BADRINARAYANAN *ET AL.*'S SEGNET ARCHITECTURE

The SegNet architecture strongly resembles that of DeconvNet and is also aimed at semantic segmentation. However, its authors demonstrated the need for returning to a significantly simpler architecture in order to make it more easily trainable (Badrinarayanan et al., 2015). Basically, their architecture was identical to

DeconvNet (Fig. 15.13) but with F6 and F7 excluded. In addition, it was clear to the authors that use of max-pooling and subsampling reduce feature map resolution and thereby reduce location accuracy in the final segmented images. Nevertheless, they start by eliminating the fully connected layers of VGGNet, retain the encoding−decoding (CNN−DNN) structure of DeconvNet, and also retain max-pooling and unpooling. In fact, it is the move away from using of fully connected layers that helps SegNet the most, as this drastically reduces the number of parameters to be learnt (see Table 15.5), and thereby also drastically reduces the training requirements of the method. Accordingly, the whole network can be considered a single rather than a dual network and trained efficiently "end-to-end." Furthermore, the authors identified a far more efficient way to store object location information: they do so by storing *only* the max-pooling indices, viz. the locations of the maximum feature values in each pooling window in each encoder feature map. As a result only 2 bits of information are needed for each 2×2 pooling window (cf. Fig. 15.12). This means that even for the initial CNN (encoder) layers, it is not necessary to store the feature maps themselves: what has to be stored is the object location information. By this means the encoder storage requirement is reduced from 134M (corresponding to layers C1−F7 of VGGNet in Table 15.5) down to 14.7M—or only a small fraction of this if it were recoded as 2 bits (instead of two floating point numbers) per pooling window. The total storage for SegNet is rated at twice this, as the same amount of information has to be saved in the decoder layers. However, the same applies to other deconvnets, so in all cases the total amount of data has to be doubled relative to the contents of the initial CNN encoder.

The smaller size of SegNet makes end-to-end training possible, and thereby far more suitable for real-time applications. The authors acknowledge that larger networks can work better—though at the cost of far more complex training procedures, increased memory, and considerably increased inference time. Furthermore, it is difficult to assess their true performance. Basically, the decoders have to be trained via very large and cumbersome encoders, and the latter are general purpose rather than being targeted at specific applications (note that the amount of effort required to train such encoders is so great as to discourage workers from retraining them to adapt them to their own applications). In the majority of instances, such networks have been based on a VGGNet front end, typically containing all 13 sublayers of C1−C5, together with a variable (very small) number of fully connected layers.

These considerations make it no surprise that Badrinarayanan et al. successfully applied SegNet to the CamVid dataset (Brostow et al., 2009) by training it end-to-end for optimum adaptation. They found it outperformed seven conventional (nonneural) methods, including local label descriptors and superparsing (Yang et al., 2012; Tighe and Lazebnik, 2013), obtaining scores averaging to 80.1%, in comparison with 51.2% and 62.0%, respectively; the 11 categories to be recognized were building, tree, sky, vehicle, sign, road, pedestrian, fence, pole, pavement, and cyclist, and the accuracies attained for these ranged from 52.9%

(cyclist) to 94.7% (pavement). Their success with this task may be judged from the results of their online demo (http://mi.eng.cam.ac.uk/projects/segnet/—website accessed 7 October 2016), which was used to generate the pictures in Fig. 15.14: In fact, this demo placed pixels in 12 categories, including road markings in addition to the 11 listed above: see Fig. 15.14.

FIGURE 15.14

Three road scenes taken from the front passenger seat. In each case, the image on the left is the original, and the image on the right is the segmentation produced by SegNet. The key indicates the 12 possible meanings assigned by SegNet. Although location accuracy is not perfect, the assigned meanings are generally reasonable, given the limited number of allowed interpretations and the variety of objects within the fields of view. These pictures were processed using the online demo at http://mi.eng.cam.ac.uk/projects/segnet/ (Badrinarayanan et al., 2015).

They also did a careful comparison of SegNet with other recent semantic segmentation networks, including FCN and DeconvNet. FCN and DeconvNet have the same encoder size (134M); note that FCN reduces the decoder size down to 0.5M, though DeconvNet continues with a 134M decoder. Class averages for the three methods are 59.1%, 62.2%, and 69.6%, making SegNet the numerically worst of the three, though its accuracy is still competitive, and it has the distinct advantage of being more adaptable by virtue of being trained end-to-end. In fact, it is also easily the fastest running, being $\sim$2.2 times faster than FCN and $\sim$3.3 times faster than DeconvNet, albeit (because of availability of results) on different sized images.

Overall, the authors state that architectures "which store the encoder network feature maps in full perform best but consume more memory during inference time," which also means that they run more slowly. On the other hand, SegNet is more efficient as it only stores the max-pooling indices; in addition, it has competitive accuracy, and its capability for end-to-end training on currently relevant data make it significantly more adaptable.

15.11 RECURRENT NEURAL NETWORKS

No sooner had we embarked upon this chapter on deep-learning networks than we realized that the most obvious upgrade path from traditional ANNs was that of using convolutional neural elements. No other possibility seemed to arise for modifying or upgrading the architecture. In retrospect, that was largely because we were primarily intent on recognition—whether of single objects or in the end for performing semantic segmentation of whole images. But while we had a firm mind-set for analyzing static images, in practice, we also need to have our eyes set on video interpretation. In fact, image sequences are best and most compactly described by telling the story underlying any video. The next step is to find how best to manage the time element. To achieve this, it is natural to use a state machine approach, and a simple way forward is to feed the outputs of the network back into its own inputs—thereby giving rise to a RNN.

It will be recalled that both ANNs and CNNs were structured carefully into layers, so that each output could only be fed to the inputs of the *following* layer. We now generalize this, so that each neuron output can be fed to at least one input on each neuron. As a result, not only will each neuron be looking at normal, current input signals, but also it will emit signals that reflect the history of the network. Note also that the feedback that is now present has the capability of making the neurons act like flip-flops containing memory, so the entire network will react with strongly time-dependent behavior. Furthermore, although a network of this type is used for testing, it will automatically be subject to continuous training, and the result will also be strongly order-dependent.

In fact, the above description of the new RNN is rather too unconstrained and general: it is better to think of its structure unfolding along the time dimension (Fig. 15.15) and to interpret its operation as performing the same task for every element of an image sequence. Notice that, just as the CNN neurons in each layer performed exactly the same operation at every neuron in a layer, so now the RNN

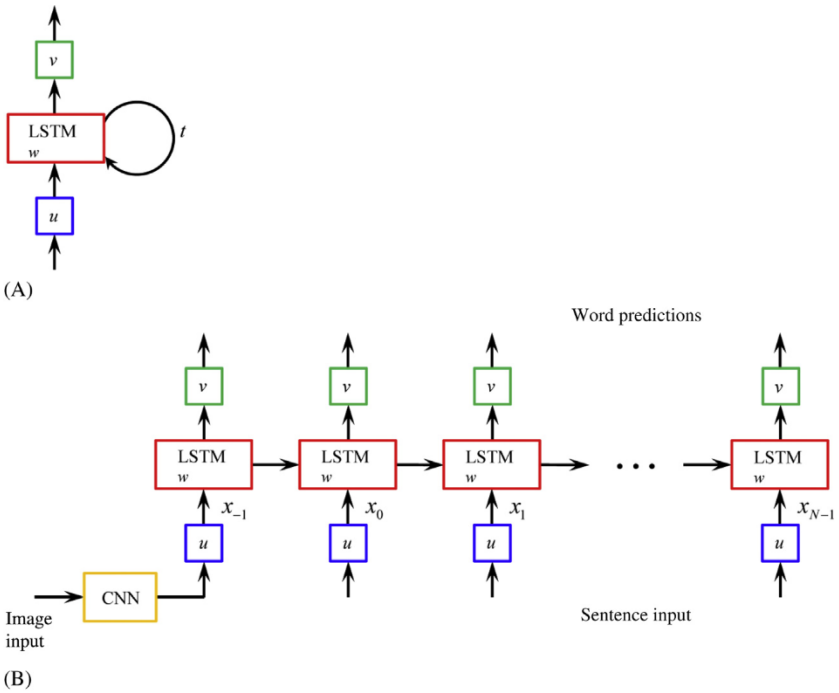


FIGURE 15.15

Recurrent neural networks and their application. Part (A) shows how a neuron (here marked "LSTM": see text) is employed in a recurrent neural network: a single feedback loop has been connected between an output and an input in order to allow the neuron to undergo temporal development (as indicated by t). Part (B) shows how the feedback loop can be "unfolded," to clarify how the network operates after initialization at $t = -1$. Note that the same parameters u, v, and w apply at each instant of time: u and v are multiplicative parameters that are applied at each input and output, whereas w is the weight of the neuron, which is applied internally. In fact, (B) shows more than just the temporal development of the neuron in (A): it actually shows the caption-generating system of Vinyals et al. (2015). In particular, it shows how an image is fed to the RNN via a CNN which extracts and identifies various image features, and these are then analyzed by the RNN to give word predictions. To achieve this, N sentence inputs [Basically, the system is trained using images accompanied by word inputs—though these can be "sentence" inputs consisting of word vectors (see Mikolov et al., 2013).] are also provided after $t = -1$, i.e., for $t = 0, \ldots, N - 1$. In fact, the word applied at $t = 0$ is the "start" signal and that applied at $t = N$ is the "stop" signal.

will perform exactly the same operation at every time element. In practice, this means that the inputs are all equally weighted as u, the outputs are all equally weighted as v, and the multiplicative weights in each neuron are all equally weighted as w.

So far, we have not shown how training will proceed, but this is simply achieved by replacing the backpropagation algorithm by an exactly analogous backpropagation through time (BPTT) algorithm. The overall architecture can now be imagined as a set of identical copies of the same network, with each stage passing data to its temporal successor. There is also the possibility of making the network bidirectional, so that data can be passed from the future back to the past. Stated in this way, the concept makes little sense, but it is actually a useful recipe when dealing with natural language, as sentences have to make global sense, and each word has to be a good contextual fit for its location in a sentence. Moreover, if image sequences are to be interpreted correctly, the output sentence descriptions will also have to make global sense. For clarity, and in spite of its recent popularity, we shall ignore the bidirectional possibility in what follows.

There is one particular difficulty that RNNs have been subject to that did not arise with ANNs or CNNs: whereas the latter could be imagined as completely analogue in operation, with each set of neuron inputs immediately giving rise to a corresponding output, and each set of output signals quickly rippling through the entire machine, with an RNN this process would lead to "race" conditions and it would not be clear how the old memory states would propagate through the system and whether they would result in robust, reliable stored memories, and output signals. These considerations led to neurons having to be very carefully designed with three logistic gates being included inside each neuron. The resulting neurons were called LSTM units. These incorporate safeguards to make the whole network operate reliably, and they have been almost universally employed ever since they were invented by Hochreiter and Schmidhuber in 1997. It should be added that part of their function is to arrange forgetting of old, now irrelevant data; to terminate complete sentences; and to start new ones when appropriate. Finally, note that in many texts, the failure modes of RNNs were originally described as vanishing and exploding gradients (which are particularly serious for the BPTT algorithm) and arose as the analog signals they represented were not properly controlled. The reason for using long short-term memories (LSTMs) was to allow the network parameters to learn reliably and consistently over many time steps.

Considerable work has now been carried out using RNNs. Amongst the most important topics in this area has been that of automatic annotation of images and videos. Notable amongst the most recent papers is that of Vinyals et al. (2015) describing their automatic image caption generating system. This carefully maps both the image and the words into the same space, with the image entering the vision CNN and the words entering as a word embedding into the RNN (Fig. 15.15B). Interestingly, the image is only fed to the first LSTM input: indeed, Vinyals et al. found that feeding the image in repeatedly (i.e., at each time step) let to inferior results as image noise then made the network more prone to overfitting. Overall, the network is trained to minimize the sum of the negative log

likelihood of the correct word being used at each time step, i.e., to ensure that the sentence (sequence of words) that is output for each image is the one with maximum likelihood of being correct. Typical captions include "A group of young people playing a game of frisbee" and "A herd of elephants walking across a dry grass field"—which, in the examples given (Vinyals et al., 2015), were not only correct descriptions of actual video actions but also quite perceptive comments. It is salutary to note that in this work, the RNN is essentially performing a translation function—in this case between the internal codes issuing from the CNN classifier output and the real (English) words the RNN is trained on.

In other work (Vondrick et al., 2016), a relatively simple RNN using an AlexNet CNN front end was trained to make predictions 1 second before an event (e.g., whether two people would hug, kiss, or "high-five") that were at least 19% more accurate than for previous methods—albeit the absolute performance was some way below that of human operators.

15.12 CONCLUDING REMARKS

This has been an unusual chapter, in that about 80% of the material has only been published in the last 4−5 years—though of course that material rested strongly on foundations that had been developing over the previous quarter century. The hugely accelerating progress over these past few years is evidenced—unusually for a subject like computer vision—by a considerable proportion of the key papers being made available first on arXiv—a feature that has hitherto been more familiar in particle physics and cosmology than in computer vision. All this created difficulties in gathering together the vital information and in presenting it in any ordered way. In fact, it presented the need for writing it as a series of key case studies, and those that were selected for this purpose were made to tell a developmental story about the subject. However, there is a pitfall in this approach, as it risked putting together a hotchpotch of facts and of claims and counterclaims, but without providing sufficient guidance for the reader on the underlying science or indeed the logic underlying all the work. Therefore, care had to be taken to allow these aspects to emerge. As a result, some of the case study descriptions turned out to be longer than expected, though in the end, this is all to the good as underlying explanations are the nonnegotiable part of any book: thus, the case studies have to be taken as vehicles upon which the principles and explanations of scientific detail are carried along. Then, there are the many terms that are passed around in the research literature, but which one often finds are undefined—"everyone" knows what they mean but in many cases they are never clearly written down, at least in citable places. Indeed, one suspects that they are often handed down solely by word of mouth at conferences and private meetings. As far as possible, to provide a logical development, all such terms have been sought out and defined as and where necessary.

The account starts by describing CNNs and how they are put together—this being followed immediately by an explanation of CNN architectures and the relevant technical terms for describing them: these include terms such as depth, stride, zero padding, receptive field, pooling, and ReLU. But of course, the main important question at this early stage is why convolutional networks won out: in short, this was because of the insistence on position invariance in each layer, which in turn limited the number of network parameters to be learnt by training. Understanding of all these considerations allowed us to go on to study LeCun et al.'s LeNet architecture: this originated in the 1990s and culminated in their key paper in 1998. Although this architecture remained a paradigm approach, it was not especially successful vis-à-vis other nonneural approaches. It was only in 2012, when it was radically updated in various key ways by Krizhevsky et al. in the form of AlexNet, that this approach really took off, and was shown to be superior to previous nonneural approaches. Hence, AlexNet formed the basis for the next case study, which was pursued at length in Section 15.5. In fact, it wasn't merely the architecture that was original but also the methods for generating and augmenting training sets: it had become clear when training huge networks containing $\sim$650,000 neurons with $\sim$60 million trainable parameters, that a mere million images (1000 from each of 1000 classes) is wildly insufficient. And so additional images were obtained (1) by extracting numerous patches and expanding numbers of examples by applying horizontal reflections and (2) by applying transformations to the intensities and colors in order to further increase the quantity of valid training samples—both of these strategies proving successful.

Almost before the ink had dried on the AlexNet paper, Zeiler and Fergus (2014) found how to optimize the architecture further and produced means for visualizing its operation—on the basis that understanding its modus operandi would lead to further improvement. This did turn out to be possible, and so-called ablation studies helped further. (At this point, ablation was another technical term that had to be explored and explained.)

Zeiler and Fergus's ideas on deconvolution for analyzing the internal operation of a CNN proved vital to others for producing the new deconv networks which would not only allow objects to be classified, but also by inverting the process, to allow the classes to be transformed into pixel maps, thereby leading to semantic segmentation of the original images. In this area, the works of Noh et al. (2015) and Badrinarayanan et al. (2015) were seminal—though others also made valuable contributions. However, not to be forgotten was Simonyan and Zisserman's VGGNet architecture (2015), which the latter two papers very much relied upon: this was because Simonyan and Zisserman found how to make CNNs of the AlexNet ilk significantly deeper (theirs was the first to be reasonably termed "very deep")—by various means, including in particular that of narrowing the receptive fields of the convolution kernels down to a maximum of 3×3.

All this was a long story—and future work will show that it is still far from being complete. Indeed, even if the VGGNet architecture is near the optimum that is possible for a multioutput classifier that is fed with raw images, it remains to be seen how well it can be applied for similar applications in acoustics and seismology

or in 3-D applications; and how widely it can be used in more general situations than those of classification and semantic segmentation. For example—as outlined in Section 15.11—Google have published a paper (Vinyals et al., 2015) in which they show nontrivial pictures that have been described by their automatic image-caption-generating system: when sufficiently developed, this will be of distinct value for searching the web for specific types of picture. Even at this stage, the work involves sentence generation combined with deep neural analysis of images and will have to be watched closely for further progress. This sort of work will clearly be of value in talking to and controlling robots. However, here, we avoid the temptation to include excessive speculation: our aim is merely to show what has so far been achieved and to hint at what might be possible. In any case, even if the basic theory of deep neural networks progresses little further for some time, maximizing the impact of what has been done so far and advancing the methodology will best be achieved by wide exploration of potential applications.

Finally, perhaps the most lasting impact of CNNs such as VGGNet is that of providing highly effective pretrained feature spaces for other classifiers, to which SVM and other networks can readily be added to tackle and solve new tasks. As has already been remarked, VGGNet has been the preferred choice in the vision community for performing this function, but it should be emphasized that it has not been alone: e.g., GoogLeNet has also been applied in this way. In a case in point, Bejiga et al. (2017) have described how they combined a pretrained GoogLeNet front end with a linear SVM classifier to assist avalanche search and rescue operations by analyzing unmanned aerial vehicle (UAV) image sequences. Interestingly, their combined CNN−SVM classifier easily outperformed a traditional HOG−SVM classifier on two datasets and resulted in accuracies well in excess of 90% for several videos. In another case, Ravanbakhsh et al (2015) used an AlexNet front end with an SVM classifier for performing human action recognition, again attaining performance levels well above those of traditional methods when applied to sports and other videos.

This chapter has covered deep-learning networks and their application to classification and segmentation. Their explosive development in the 2010s has been highlighted, and the ground rules for deep-learning architectures have been explained at some length with the aid of a number of key case studies. What happened in 2012 was the sudden realization that deep neural networks are now superior in performance to even the best conventional (nonneural) approaches in a handful of important application areas. They also seem to be making valuable progress in other areas such as automatic image caption generation.

15.13 BIBLIOGRAPHICAL AND HISTORICAL NOTES

This rather unusual chapter reflects a slow systematic advance in the development and use of ANNs—during which CNNs gradually came to dominate the scene—

this being followed by an explosive series of advances which took place from 2011 onward. The key event that initiated the latter change was the attempt by Krizhevsky et al. to respond to the ILSVRC in 2012. After augmenting the by then standard LeCun et al. (1998) LeNet architecture, and making various what might in 2011−12 have been classed as 'daring' moves—including use of ReLUs, together with overlapping (max-) pooling, huge numbers of layers, and use of dropout—they implemented the whole machine as a dual GPU architecture joined only at a limited number of layers; finally, they trained the system not only on the 1.2 million images supplied for the challenge but in the end $\sim$2000 times that number. To achieve all that within a year was a formidable task, bearing in mind that it was not known in advance that these measures would actually work in practice. Needless to say, in the following year or two, many other workers followed this advance with improvements of their own—notable amongst these being Simonyan and Zisserman (2015), who found how to make networks "very deep"—in particular, by narrowing the receptive fields of the convolution kernels to a *maximum* of 3×3. A notable application of this type of work was that of Badrinarayanan et al. (2015), which was aimed at semantic segmentation.

At this point, the theoretical development of the CNN approach seemed to slow down, and attention turned to finding what applications could be dealt with using existing theory and the experimental methods so far developed. This meant that large numbers of papers suddenly started to be produced in various application areas. Notably, these included face detection and recognition. Developments in the latter areas will be covered more fully in Chapter 21, Face Detection and Recognition: the Impact of Deep Learning. However, the work done by Taigman et al. (2014) is worth recalling: their "DeepFace" approach to face recognition brought the performance level up to 97.35%, which was extremely close to that for human recognition. Interestingly, they achieved this using an initial "frontalization" technique aimed at standardizing faces to a symmetric frontal view. Sagonas et al. (2015) developed their own frontalization technique obtaining that can best be described as a trained eigenset of frontal images. At about the same time, Yang et al. (2015, 2017) developed a high-performance Faceness-Net face detector using a CNN architecture for finding attributes of face images. This was run forwards to find the intrinsic face features and "in reverse" to regenerate localized face-part response maps: here, they followed the coding−decoding procedures set out by Zeiler and Fergus (2014). Finally, Bai et al. (2016) produced a very much simpler and faster design: this is a FCN working at multiple scales, with five shared convolutional layers followed by a branching out into two further convolutional layers—the latter respectively coping with (1) the multiple scales and (2) the sliding window effect needed to perform the final matching. An interesting and actually quite powerful conclusion is that a simpler system that is trained end-to-end has a better chance of achieving superior performance than one that is put together using separately pretrained sections.